Optimization Techniques for Distributed Task-based Programming Models

Omar Shaaban Ibrahim Ali

Department of Computer Architecture
Universitat Politècnica de Catalunya

This dissertation is submitted for the degree of
Doctor of Philosophy

Barcelona, July 2024

Optimization Techniques for Distributed Task-based Programming Models

by

Omar Shaaban Ibrahim Ali

A Dissertation
Presented to the Department of Computer Architecture
at
Universitat Politècnica de Catalunya
in Candidacy for the Degree of
Doctor of Philosophy.

Thesis Advisors:

Paul Carpenter
Barcelona Supercomputing Center, Spain

Prof. Xavier Martorell Bofill
Universitat Politècnica de Catalunya, Spain

Barcelona, July 2024

Optimization Techniques for Distributed Task-based Programming Models

Abstract

As High Performance Computing (HPC) hardware technology evolves rapidly, it is essential for applications to be able to fully leverage the potential performance and capabilities of a diverse range of hardware and requirements. Most HPC applications employ the Message Passing Interface (MPI) model, which requires the explicit specification of communication, making the development process error-prone and cumbersome. Even worse, combining MPI with shared memory models such as OpenMP and OmpSs-2 increases complexity and raises the risk of deadlock.

In HPC, task-based models have gained prominence via the adoption of tasks in OpenMP, as an asynchronous and platform-agnostic high-performance and productive model by annotating existing code, transforming it into a parallel version. A program is expressed as a Directed Acyclic Graph (DAG), whose vertices are units of code called tasks. Edges between tasks represent dependencies between them, and tasks with no logical relationship can be executed concurrently. The task graph is independent of the targeted platform architecture, making these models suitable for concurrent execution on a broad spectrum of platforms such as multi-core SMPs or offloaded to GPUs, FPGAs, or accelerators.

Several initiatives explore distributed-tasking approaches, which use the task model to decompose the application across multiple nodes with distributed memory. The advantage is that the application is expressed in a simple and clean manner that reflects only the computations and dependencies. Unfortunately, the distributed tasking approaches suffer from poor efficiency and scalability, which hinder their adoption by the HPC community. This is mainly due to the overhead of task creation and dependency graph construction, which are usually sequential. This thesis proposes two techniques to address this problem.

Our first approach relates to task nesting, which mitigates the sequential bottleneck by building the full dependency graph in parallel using multiple concurrently executing parent tasks. A key limitation of task nesting is that a task cannot be created until all its accesses and its descendants' accesses are known. Current approaches to work around this limitation either halt task creation and execution using an explicit `taskwait` barrier or substitute dependencies with artificial accesses known as sentinels. We introduce the `auto` clause, which indicates that the task may create subtasks that access unspecified memory regions, or it may allocate and return memory at addresses that are not yet known. Contrary to `taskwait`, our approach does not prevent the concurrent creation and execution of tasks, maintaining parallelism and allowing the scheduler to optimize load balance and data locality. In addition, all tasks can be given a precise specification of their own data accesses, unlike sentinels, resulting in a unified mechanism governing task ordering, program data transfers on distributed memory, and optimizing data locality, e.g. on NUMA systems. The `auto` clause, therefore, provides an incremental path to develop programs with nested tasks by removing the need for every parent task to have a complete specification of the accesses of its descendent tasks while reducing redundant information that can be time-consuming and error-prone to describe.

Our second approach takes advantage of the iterative behaviour of many HPC applications, such as those that employ iterative methods or multi-step simulations. Most models construct the full unrolled task graph sequentially despite the fact that these applications create the same directed acyclic graph of tasks on each timestep. We define the programming model based on the `taskiter` clause, a recently introduced construct in the literature for iterative applications on SMP [1]. We also describe the full runtime implementation to exploit this information to eliminate the sequential bottleneck and control messages while retaining the simplicity and productivity of the existing approach.

In summary, this thesis makes two advances towards making the distributed tasking approach a viable alternative to MPI + OpenMP. For applications without an iterative structure, the `auto` keyword simplifies the use of task nesting, which is the most powerful way to mitigate the sequential bottleneck. For applications with an iterative structure, the taskiter approach amortizes the control overhead across all loop iterations, enabling a performance that matches the best MPI-based approach. We integrate both techniques into OmpSs-2@Cluster, the distributed tasking variant of OmpSs-2, and evaluate the performance on the MareNostrum 4 supercomputer.

Keywords: HPC, Runtime Systems, OpenMP, OmpSs-2, Distributed computing, Task Parallelism.

Resumen

A medida que la tecnología de hardware para la Computación de Alto Rendimiento (HPC) evoluciona rápidamente, es fundamental que las aplicaciones puedan aprovechar plenamente el rendimiento y las capacidades potenciales de diversas plataformas de hardware y requisitos. La mayoría de aplicaciones HPC utilizan el modelo de Interfaz de Paso de Mensajes (MPI), que exige la especificación explícita de la comunicación, convirtiendo el proceso de desarrollo en un reto propenso a errores y laborioso. Aún más, la combinación de MPI con modelos de memoria compartida como OpenMP y OmpSs-2 incrementa significativamente la complejidad y el riesgo de bloqueo mutuo.

En HPC, los modelos basados en tareas han ganado prominencia adoptando OpenMP como un modelo de alto rendimiento, productivo, asincrónico y agnóstico respecto a la plataforma dentro del nodo. Un programa se expresa como un Grafo Acíclico Dirigido (DAG), cuyos vértices son unidades de código llamadas tareas. Los arcos entre las tareas representan dependencias entre ellas, y las tareas sin relaciones lógicas pueden ejecutarse concurrentemente. El grafo de tareas es independiente de la arquitectura de la plataforma objetivo, haciendo estos modelos adecuados para la ejecución concurrente en una amplia gama de plataformas, incluyendo multiprocesadores y aceleradores.

Diversas iniciativas están explorando enfoques de tareas distribuidas, que utilizan el modelo de tarea para descomponer la aplicación en varios nodos con memoria distribuida. La ventaja de este enfoque es que permite expresar la aplicación de manera simple y clara, reflejando únicamente los cálculos y las dependencias. Desafortunadamente, las implementaciones actuales de tareas distribuidas sufren de baja eficiencia y escalabilidad, lo que dificulta su adopción por parte de la comunidad de HPC. Esto se debe principalmente al alto costo de creación de tareas y a la construcción del grafo de dependencias, que generalmente son secuenciales. Esta tesis propone dos técnicas para solucionar este problema.

Nuestra primera propuesta se centra en la anidación de tareas, que atenúa el cuello de botella secuencial mediante la construcción paralela del grafo completo de

dependencias usando múltiples tareas maestras que se ejecutan concurrentemente. Una limitación clave de la anidación de tareas es que no se puede crear una tarea hasta que se conozcan todos los accesos de memoria de la misma y de sus descendientes. Los enfoques actuales para resolver esta limitación detienen la creación y ejecución de tareas utilizando una barrera explícita (taskwait) o sustituyendo las dependencias con accesos artificiales llamados centinelas. Proponemos la cláusula auto, que indica que la tarea puede crear subtareas que accedan a regiones de memoria no especificadas, o puede asignar y devolver memoria en direcciones aún desconocidas. A diferencia de una barrera explícita (taskwait), nuestro enfoque no impide la creación y ejecución concurrente de tareas, manteniendo el paralelismo y permitiendo al programador optimizar el balance de carga y la localización de datos. Además, todas las tareas pueden tener una especificación precisa de sus propios accesos a datos, a diferencia de los centinelas, resultando en un mecanismo unificado que gobierna el orden de las tareas, las transferencias de datos del programa en memoria distribuida y la optimización de la localización de datos, por ejemplo en sistemas NUMA. Así, la cláusula auto proporciona un camino incremental para el desarrollo de programas de tareas anidadas, eliminando la necesidad de que cada tarea padre tenga una especificación completa de los accesos de sus tareas descendientes, reduciendo a su vez la información redundante que puede ser tediosa y propensa a errores.

Nuestra segunda propuesta aprovecha el comportamiento iterativo de muchas aplicaciones HPC, como aquellas que utilizan métodos iterativos o simulaciones en múltiples etapas. A pesar de que estas aplicaciones generan el mismo grafo acíclico dirigido de tareas en cada paso temporal, la mayoría de los modelos construyen el grafo completo de tareas de manera secuencial. Definimos el modelo de programación basado en la cláusula taskiter, una propuesta recientemente introducida en la literatura para aplicaciones iterativas en plataformas SMP. También describimos la implementación completa en tiempo de ejecución para aprovechar esta información y eliminar el cuello de botella secuencial y los mensajes de control, preservando la simplicidad y productividad del enfoque existente.

Integramos ambas técnicas en OmpSs-2@Cluster, la variante de tareas distribuidas de OmpSs-2, y evaluamos el rendimiento en el superordenador MareNostrum 4.

En resumen, esta tesis realiza dos avances para mejorar la viabilidad de los modelos de tareas distribuidas como una alternativa a MPI + OpenMP. Para aplicaciones sin estructura iterativa, la clave 'auto' simplifica el uso de la anidación de tareas, que es la forma más eficaz de mitigar el cuello de botella secuencial. En el caso de aplicaciones con estructura iterativa, la propuesta basada en el 'taskiter' distribuye la sobrecarga de control a través de todas las iteraciones del bucle, posibilitando un rendimiento equiparable a la mejor aproximación basada en MPI. Los trabajos futuros pueden continuar desarrollando extensiones a partir de nuestras propuestas.

Resum

A mesura que la tecnologia de maquinari per a la Computació d'Altes Prestacions (HPC) evoluciona ràpidament, és fonamental que les aplicacions puguin aprofitar plenament el rendiment i les capacitats potencials de diverses plataformes de maquinari i requisits. La majoria d'aplicacions HPC utilitzen el model d'Interfície de Pas de Missatges (MPI), que exigeix l'especificació explícita de la comunicació, convertint el procés de desenvolupament en un repte propens a errors i feixuc. Encara més, la combinació de MPI amb models de memòria compartida com OpenMP i OmpSs-2 incrementa significativament la complexitat i els risc de bloqueig mutu.

En HPC, els models basats en tasques han guanyat prominència adoptant OpenMP com a model d'alt rendiment, productiu, asincrònic i agnòstic respecte a la plataforma dins del node. Un programa s'expressa com un Graf Acíclic Dirigit (DAG), els vèrtexs del qual són unitats de codi anomenades tasques. Les arestes entre les tasques representen dependències entre elles, i les tasques sense relacions lògiques poden executar-se concurrentment. El graf de tasques és independent de l'arquitectura de la plataforma objectiu, fent aquests models adequats per a l'execució concurrent en una àmplia gamma de plataformes, incloent multiprocessadors y acceleradors.

Diverses iniciatives estan explorant enfocaments de tasques distribuïdes, que utilitzen el model de tasca per descompondre l'aplicació en diversos nodes amb memòria distribuïda. L'avantatge d'aquest enfocament és que permet expressar l'aplicació de manera simple i clara, reflectint únicament els càlculs i les dependències. Malauradament, les implementacions actuals de tasques distribuïdes pateixen de baixa eficiència i escalabilitat, fet que dificulta la seva adopció per part de la comunitat de HPC. Això es deu principalment a l'alt cost de creació de tasques i a la construcció del graf de dependències, que generalment són seqüencials. Aquesta tesi proposa dues tècniques per solucionar aquest problema.

La nostra primera proposta es centra en la nidificació de tasques, que atenua el coll d'ampolla seqüencial mitjançant la construcció paral · lela del graf complet de dependències usant múltiples tasques mestres que s'executen concurrentment. Una

limitació clau de la nidificació de tasques és que no es pot crear una tasca fins que no se'n coneguin tots els accessos i els dels seus descendents. Els enfocaments actuals per a resoldre aquesta limitació detenen la creació i execució de tasques fent servir una barrera explícita (taskwait) o substituint les dependències amb accessos artificials anomenats sentinelles. Proposem la clàusula auto, que indica que la tasca pot crear subtasques que accedeixin a regions de memòria no especificades, o pot assignar i retornar memòria en adreces encara desconegudes. A diferència de una barrera explicita (taskwait), el nostre enfocament no impedeix la creació i execució concurrent de tasques, mantenint el paral · lelisme i permetent al programador optimitzar el balanç de càrrega i la localització de dades. A més, totes les tasques poden tenir una especificació precisa dels seus propis accessos a dades, a diferència dels sentinelles, resulta en un mecanisme unificat que governa l'ordre de les tasques, les transferències de dades del programa en memòria distribuïda i l'optimització de la localització de dades, per exemple en sistemes NUMA. Així, la clàusula auto proporciona un camí incremental per al desenvolupament de programes de tasques niades, eliminant la necessitat que cada tasca pare tingui una especificació completa dels accessos de les seves tasques descendents, tot reduint la informació redundant que pot ser tediosa i propensa a errors.

La segona proposta nostra aprofita el comportament iteratiu de moltes aplicacions HPC, com ara aquelles que utilitzen mètodes iteratius o simulacions en múltiples etapes. Malgrat que aquestes aplicacions generen el mateix graf acíclic dirigit de tasques en cada pas temporal, la majoria de models construeixen el graf complet de tasques de manera seqüencial. Definim el model de programació basat en la clàusula taskiter, una proposta recentment introduït a la literatura per a aplicacions iteratives en plataformes SMP. També descrivim la implementació completa en temps d'execució per aprofitar aquesta informació per eliminar el coll d'ampolla seqüencial i els missatges de control, preservant la simplicitat i productivitat de l'aproximació existent. Integrem ambdues tècniques a OmpSs-2@Cluster, la variant de tasques distribuïdes de OmpSs-2, i avaluem el rendiment al superordinador MareNostrum 4.

En resum, aquesta tesi realitza dos avanços per millorar la viabilitat del models de tasques distribuïdes com una alternativa a MPI + OpenMP. Per a aplicacions sense estructura iterativa, la clau 'auto' simplifica l'ús de la nidificació de tasques, que és la forma més eficaç de mitigar el coll d'ampolla seqüencial. En el cas d'aplicacions amb estructura iterativa, la proposta basada amb el 'taskiter' reparteix la sobrecàrrega de control a través de totes les iteracions del bucle, possibilitant un rendiment equiparable a la millor aproximació basada en MPI. Els treballs futurs poden continuar desenvolupant extensions a partir d'aquest de les nostres propostes.

Contents

Abstract vii

Resumen ix

Resum xii

List of Figures xviii

List of Tables xx

Listings xxi

Acknowledgement xxiv

Acronyms xxvi

Glossary xxviii

1 Introduction 1

1.1 Task Nesting 2

1.2 Taskiter 5

1.3 Thesis Outline 7

1.4 Publication List 8

2 Background 10

2.1 OmpSs-2 Parallel Programming Model 10

2.1.1 Execution Model 11

2.1.2 Dependency Model 12

2.2 OmpSs-2@Cluster 16

2.3 Runtime Reference Implementation 18

2.3.1 Nanos6 18

2.3.2 Nanos6@Cluster 19

2.3.3 Memory Model 21

2.3.4 Tasks, Offloading and Scheduling 22

2.3.5 Control Messages 23

2.4 Taskiter 25
2.5 Task-aware MPI (TAMPI) 27

3 Related Work 29
3.1 MPI, PGAS and Hybrid Approaches 29
3.2 Distributed Tasking 30
3.2.1 Implicit task graph creation 30
3.2.2 Concurrent and duplicated task graph creation 31
3.2.3 Sequential task graph creation 32
3.3 Other Approaches 32
3.3.1 Frameworks and Libraries 32
3.3.2 Scripting and Workflows 33
3.4 Data Access Specifications 34
3.4.1 Compile-time 34
3.4.2 Run-time 34
3.4.3 Automatic 34

4 Automatic aggregation of data accesses 36
4.1 Introduction 36
4.2 Motivation 36
4.2.1 Precise specification of data accesses without taskwaits 36
4.2.2 Productivity and Incremental Path for Nested Tasks 41
4.3 Programmer's Model 42
4.3.1 Auto Access Type 42
4.3.2 None Access Type 44
4.3.3 Upgrade Rules 45
4.3.4 Fragmentation 45
4.3.5 Inheritance of auto and none regions 46
4.4 Implementation 46
4.4.1 Compiler 46
4.4.2 Runtime 47
4.5 Methodology and Benchmarks 49
4.5.1 Hardware and software platform 49
4.5.2 Benchmarks 50
4.6 Evaluation and Results 51
4.6.1 SMP Evaluation 53
4.6.2 Cluster Evaluation (auto vs. taskwait) 56
4.6.3 Cluster Evaluation (auto vs. manual) 59
4.6.4 Quantifying productivity 61
4.7 Conclusion 62

5 Distributed taskiter 63
5.1 Introduction 63
5.2 Motivation 66
5.3 Programmer's Model 70
5.4 Implementation 71
5.4.1 Compilation 71
5.4.2 Building The Taskiter Graph 71
5.4.3 Partition The Taskiter Graph 72
5.4.4 Creating The Local Taskiter Graph 73
5.4.5 Single Iteration Execution 79
5.5 Methodology and Benchmarks 80
5.6 Evaluation and Results 82
5.6.1 Strong Scalability 82
5.6.2 Iterations Count Performance 88
5.7 Initial Overhead Analysis 90
5.8 Quantifying Productivity 91
5.9 Conclusion 92

6 Conclusions and Future work 93
6.1 Future Work 95
6.1.1 auto and none clauses 95
6.1.2 Combining auto and distributed taskiter 95
6.1.3 Overhead Optimisations 96
6.1.4 Collective Communications 96
6.1.5 Dynamic Scheduling and Load Balancing 97
6.1.6 Non-constant Iterations 97
6.1.7 Loop Unrolling 97
6.1.8 Summary 97

References 98

List of Figures

2.1 OmpSs-2 dependency annotations syntax comparison 12
2.2 Ways of specifying dependencies in OmpSs-2. 14
2.3 OmpSs-2 development flow 19
2.4 OmpSs-2@Cluster architecture 20
2.5 Nanos6@Cluster local memory model 22
2.6 Nanos6@Cluster distributed memory model 23
2.7 OmpSs-2 control messages 24
2.8 OpenMP and OmpSs-2 taskiter construct syntax 25
2.9 Example OpenMP program using taskiter 26

4.1 Executiuon trace of the hypermatrix benchmark using taskwait 38
4.2 Semantics of `auto` access type 43
4.3 Semantics of `none` access type 44
4.4 Proposed OpenMP and OmpSs-2 syntax for `auto` 45
4.5 Hypermatrix benchmark Extrae/Paraver trace 54
4.6 `hypermatrix` benchmark SMP performance 55
4.7 `matmul-smp` benchmark SMP performance 55
4.8 `n-body-smp` benchmark SMP performance 56
4.9 n-body strong scalability on 1 to 32 nodes. 57
4.10 Extrae/Paraver trace of n-body benchmark on 4 nodes. 58
4.11 `multi-matvec` strong scalability on 1 to 32 nodes. 59
4.12 `jacobi` strong scalability on 1 to 32 nodes. 60
4.13 `multi-matmul` strong scalability on 1 to 32 nodes. 60
4.14 `cholesky` strong scalability on 1 to 32 nodes. 61

5.1 Control messages for OmpSs-2@Cluster iterative application. 64
5.2 Distributed taskiter control messages. 65
5.3 Distributed taskiter execution process 71

5.4 Top and Bottom map data structures. 74
5.5 Taskiter communication tasks insertion 76
5.6 Single iteration dependency graph of Listing 5.5. 78
5.7 Benchmark: multi-matvec 83
5.8 Benchmark: multi-matmul 84
5.9 Benchmark: jacobi 85
5.10 Benchmark: heat-gauss 86
5.11 Benchmark: heat-jacobi 87
5.12 Taskiter overall performance as a function of iterations count 89
5.13 Initial overhead as a function of the number of accesses. 90
5.14 Number of accesses depending on block size. 91

List of Tables

2.1 OmpSs-2 Access upgrade rules 13

4.1 OmpSs-2 extended access upgrade rules 46
4.2 Auto evaluation benchmarks 51
4.3 Productivity as a function of number of accesses 62

5.1 Taskiter evaluation benchmarks 81
5.2 Number of lines of code for the kernels of of the benchmarks. 92
5.3 Number of lines of code for the control code of the benchmarks. 92

List of Listings

2.1 Source code of Figure 2.7 24
2.2 Example OpenMP program using TAMPI 28
4.1 Approach with additional synchronisation (taskwait) 37
4.2 Approach with a “fake dependency” (sentinel) 39
4.3 Approach with proposed `auto` data accesses 40
4.4 Parent task requires a multi-dependency for its children 41
4.5 Parent task with `auto` clause 42
4.6 Parent task with `auto` clause specifying a range 42
4.7 Example of nested tasks with weak dependencies 52
4.8 Example of `auto` clause for productivity 53
5.1 OmpSs-2@Cluster iterative version of example in Listing 2.1. 63
5.2 Fork–join MPI + OpenMP tasks of Gauss–Seidel 2D heat equation. . . 67
5.3 Asynchronous TAMPI + OpenMP/OmpSs-2 of Gauss–Seidel 2D heat equation. 69
5.4 OmpSs-2@Cluster taskiter of Gauss–Seidel 2D heat equation. 70
5.5 OmpSs-2@Cluster distributed taskiter example. 72

To the memory of my dear father:
You forever encouraged me, believed in me, and here I am now, where my ambitions meet your high hopes for me.

To mom, sister, and brother:
Your prayers and wishes reached me thousands of kilometres away. You gave me reason and purpose. Thank you.

Palestine, you made us all free

Acknowledgements

In the name of ALLAH, the most beneficent, the most merciful, the omniscient, all praises to ALLAH for the strength and insight into finishing this work.

To Paul Carpenter, my sincerest gratitude. I am fortunate to have you as my mentor, supervisor, and teacher. This PhD would have never been significant without your guidance, constant encouragement, and unconditional support. Thank you for being patient and considerate and for the long and late pair-debugging sessions. Paul is a good listener; he knows how to shape words into excellent articles, and he is thoughtful when it comes to others making mistakes.

To my thesis tutor, prof. Xavier Martorell, for his humility and immense knowledge, thank you for all the support. To Vicenç Beltran, I deeply appreciate your valuable ideas and suggestions throughout my entire PhD journey.

To my cluster team, Isabel Piedrahita and Juliette Fournis, thank you for all the meeting time I borrowed from you; I wish you a strong will (and scalability) finishing your degrees.

To all the reviewers of my thesis, internally: Daniel Jimenez, Jorge Ejarque, and Sergio Iserte, and externally: Prof. Guido Araujo and Prof. Mark Bull (in alphabetical order), thank you for your incisive and constructive comments and suggestions. It was valuable and contributed greatly to my work.

To Petar Radojkovic, thank you for ensuring we checked on life daily and for the chocolate as well. To the memory team: Mariana Carmin, Pouya Esmaili, Valéria Soldera, and Victor Xirau, thank you for expanding and taking our desks. I wouldn't have had better friends, colleagues, and companionship.

To Jimmy Aguilar, Chenle Yu, Peini Liu, and Felippe Zacarias, your company and long chats were an indispensable part of my journey here. Thanks to friends that turned into family, it was all fun and great.

My thanks are extended to Antoni Navarro and Kevin Sala for all the help. Your time was valuable and much appreciated.

To BSC, thank you all.

This research has received funding from the European Union's Horizon 2020/EuroHPC research and innovation programme under grant agreement No 754337 (EuroEXA), LEGaTO (780681), and 955606 (DEEP-SEA). It is supported by the Spanish State Research Agency - Ministry of Science and Innovation (contract PID2019-107255GB-C21/MCIN/AEI/10.13039/501100011033 and Ramon y Cajal fellowship RYC2018-025628-I /MCIN/AEI/10.13039/501100011033 and by "ESF Investing in your future"), as well as by the Generalitat de Catalunya (2017-SGR-1414).

Acronyms

API Application Programming Interface.

BLAS Basic Linear Algebra Subprograms.

BSC Barcelona Supercomputing Center.

CUDA Compute Unified Device Architecture.

DAG Directed Acyclic Graph.

DCTG Directed Cyclic Task Graph.

DLB Dynamic Load Balancing.

DTD Dynamic Task Discovery.

FPGA Field Programmable Gate Array.

GCC GNU Compiler Collection.

GPU Graphic Processor Unit.

HFI Host Fabric Adapter.

HPC High Performance Computing.

MKL Math Kernel Library.

MN4 MareNostrum 4.

MPI Message Passing Interface.

NUMA Non Uniform Memory Access.

OpenMP Open Multi-Processing. See *Glossary:* OpenMP.

PCIe Peripheral Component Interface Express.

PGAS Partitioned Global Address Space.

PMPI Message Passing Interface (MPI) Profiling Interface.

PTG Parameterized Task Graph.

SMP Shared Memory Multiprocessor System.

StarSs Star Superscalar.

STF Sequential Task Flow.

STG Sequential Task Graph.

STL Standard Template Library.

TAMPI Task-Aware MPI.

TOML Tom's Obvious, Minimal Language. See *Glossary:* TOML.

WaW Write-after-Write.

Glossary

Extrae Barcelona Supercomputing Center (BSC)'s package for generation of Paraver trace files for post-mortem analysis first.

LLVM Collection of modular and reusable compiler and toolchain technologies formerly known as the Low Level Virtual Machine.

Mercurium BSC's Source-to-source compiler used with Nanos++ or Nanos6 to implement OpenMP and OmpSs/OmpSs-2.

Nanos++ Runtime system that implements the OmpSs-1 programming model.

Nanos6 Runtime system that implements the OmpSs-2 programming model.

Nanos6@Cluster OmpSs-2@Cluster programming model reference implementation.

OmpSs OpenMP Superscalar task-based parallel programming model.

OmpSs-2 The second generation of the OmpSs task-based parallel programming model.

OmpSs-2@Cluster The distributed memory tasking model extention of the OmpSs-2 task-based parallel programming model.

OpenMP Application Programming Interface (API) for multi-platform Shared Memory Multiprocessor System (SMP) programming in C, C++ and Fortran.

Paraver BSC's parallel program trace visualisation and analysis tool.

strong access An access that is not a weak access.

TOML File format used for configuration files.

weak access An access that defines a region of memory that is only accessed by subtasks of the annotated task. A weak access acts as a linking point between dependency domains, but it does not enforce task ordering or in itself require data transfers.

weak task A task all of whose accesses are weak.

CHAPTER 1

Introduction

The dominant development approaches in High Performance Computing (HPC) leverage Message Passing Interface (MPI) + X models, where X denotes a model for shared memory parallelisation inside a node (e.g. OpenMP [2]). The rapid-progressive nature of HPC, regarding the underlying hardware and workload, mandates that applications development also matches this change. Applications that utilise MPI require explicitly specifying the communications that demand meticulous delineation of sent/received data. This inherent requirement amplifies the susceptibility to errors and introduces intricacy in the development workflow. Furthermore, integrating MPI with shared memory models like OpenMP and OmpSs-2 exacerbates the complexity, increasing the likelihood of deadlock.

Led by the massive adoption of OpenMP in HPC [3], task-based models stood out to fit these applications' requirements as asynchronous and platform-agnostic high-performance and productive models. Programs are expressed in terms of units of code called tasks, which are connected together via dependencies to form a Directed Acyclic Graph (DAG). Edges between tasks represent dependencies between them, and tasks with no logical relationship can be executed concurrently. The task graph represents computations independently of the targeted platform architecture, making it suitable for concurrent execution on a broad spectrum of platforms, ranging from multi-core Shared Memory Multiprocessor Systems (SMPs), Graphic Processor Units (GPUs) [4, 5], or Field Programmable Gate Arrays (FPGAs) [6].

In addition, applications can further divide their computations and offload them to multiple cluster nodes for maximum utilisation of the targeted machine in what is known as distributed-memory tasking approaches. This has been investigated by multiple initiatives showing a promising outlook for accelerating (HPC) applications as shown in related work (Section 3) of this thesis. The single-task graph unifies the representation of parallelism across CPU cores, accelerators, and distributed-memory nodes, and it improves the productivity of MPI + X approaches.

Sequential Task Graph (STG) models, such as OpenMP and OmpSs-2, create the task graph sequentially, which provides a clear and familiar programming interface, simplifying development and maintenance and facilitating the porting of existing codes so that the developer's code reflects only what needs to be computed.

However, being prominent, distributed-tasking approaches suffer from poor efficiency and scalability [7], which pose challenges hindering their adoption by the HPC community. Primarily, these challenges stem from the sequential bottleneck for medium and fine-grained-size tasks, which limits performance and scalability. Unless the tasks are very large, distributed sequential task graph approaches are not a viable alternative to MPI + X. This thesis aims to resolve the former issues by proposing two approaches. However, each approach tackles these inefficiencies as a consequence of different case scenarios.

In the first scenario, the application task graph exhibits a hierarchical structure in which some tasks are encapsulated inside other tasks to express parallelism at its maximum degree and the reduction of synchronisation and contention that are intrinsic to the application. This is referred to as *task nesting* and tackled by approach in (Section 1.1). The second case, introduced by models that construct the task graph sequentially, imposes a performance bottleneck for medium and fine-grained size tasks (i.e. larger graphs), limiting the performance by the sequential filtering of the dependencies between tasks. This is addressed by the approach introduced in (Section 1.2).

1.1 Task Nesting

As systems scale to larger numbers of cores and accelerators, it becomes infeasible for one sequential thread to create enough tasks to keep all the cores busy. The natural solution is to build the full dependency graph in parallel by concurrently executing multiple parent tasks. This approach is known as task nesting, and it is supported by OpenMP [8, 9]. In OpenMP, the dependency graphs of different tasks are isolated. OmpSs-2 improves the situation through its support for fine-grained dependencies among nesting levels via introducing the *weak-dependencies* [10] with parent tasks having weak accesses and their subtasks have strong accesses. A weak accesses imply that the task itself does not access the data, however its subtasks are accessing it, and in this case the subtasks will have their dependencies declared as strong accesses.

The computation of the dependencies from the data accesses is done locally inside a task so that all fine-grained tasks in the program become part of a single hierarchical dataflow dependency graph. This allows dependencies to be discovered among subtasks of different parents simultaneously.

However, a key limitation of the current OpenMP and OmpSs-2 tasking models is that a task cannot be created until the addresses and sizes of all its accesses are known. Since the task's accesses must cover all its descendants' accesses, all the accesses of its subtasks, their subtasks, and so on, also need to be inside specified regions. We illustrate the problem with a hypermatrix multiplication followed by a Cholesky decomposition described in Section 4.2.1. These task accesses may be unknown prior to the task creation time because either:

1. The task allocates and returns new memory regions later in future that we do not have any information regarding it at the present moment.

2. The task creates subtasks that access memory regions (dependencies) defined by previous tasks in the application. The dependencies are known. However, these subtasks can not be created until the previous tasks are finished. Hence, we must maintain the sequential order between tasks before creating any.

3. The task accesses are complicated to determine or express, and the programmer must give a complete and precise description of the task dependencies, which in some cases might require manually computing the start and length of each access, which is an error-borne, fragile, and time-consuming step. Hence, the programmer decides not to explicitly specify all the accesses on behalf of the task and its subtasks.

Existing approaches work around this problem in different ways. One way is to add synchronisation to wait for the earlier tasks to complete, i.e., using a `taskwait` barrier. Another way is to substitute the true data access with a fake dependency that implies the correct task ordering but does not provide correct information for data transfers or data locality, i.e., using a sentinel such as a pointer to a specific memory region rather than the data itself.

Chapter 4 of this thesis introduces the `auto` data access clause that addresses the performance limitation of existing approaches. The `auto` clause indicates that a task may have additional data accesses beyond those listed explicitly in a `pragma` annotation of a task.

While the `taskwait` approach interrupts the concurrent creation and execution of tasks, our approach does not, and this maintains parallelism and provides maximum freedom to the scheduler to optimise any load balancing and data locality operations. Unlike approaches using sentinels, all tasks can be given precise specifications for strong data access. The full specification of task data accesses means that a single mechanism is used to compute the dependencies that enforce ordering among tasks, program data transfers on distributed memory, and optimise data locality on NUMA and distributed memory systems. Compared with a version using sentinels, program clarity is improved since the `pragma` annotations match the program's actual data accesses (reflects what the programmer wants to compute). It avoids the fragility of sentinels, allowing part of the program to be modified (e.g. change task granularity) without redesigning the use of sentinels throughout the whole program. By precisely specifying every task's true data access, the program is also suitable for task offloading on distributed memory systems.

The `auto` data access clause also provides an incremental path to developing programs with nested tasks. Since `auto` declares that the data access annotations may not cover all accesses of descendent tasks, it becomes only necessary to annotate the strong accesses for the task itself. For some applications, such as dense linear algebra, it may be straightforward to specify the precise weak accesses, but for applications involving graphs or trees, this task is more difficult. Moreover, these weak access annotations are redundant and error-prone.

While the implementation would be complicated and expensive in terms of overhead, we present a simple approach, with a few key optimisations, that allows efficient task execution, even in the context of task offloading to other nodes. Since the `auto pragma` aggregates information that is already given to the runtime system, there is no need for sophisticated compiler or instruction-level analysis. In many cases, the overhead is negligible, though, of course, when `auto` is used indiscriminately, performance analysis may show overhead or serialisation that can be avoided by specifying the data accesses of just a subset of the parent tasks.

1.2 Taskiter

A common approach of distributed-memory tasking as in OmpSs-2@Cluster [7], StarPU-MPI [11], PaRSEC [12], OMPC [13] and others [14] is expressing an HPC application using a single graph of tasks with dependencies. The runtime system maps the tasks to processes and executes them concurrently across the available compute nodes. This approach avoids the synchronization and deadlock issues of the more common MPI+X approach [15–17], in addition to the ability to extend them to support transparent dynamic load naturally balancing [18, 14] and both core-and node-level malleability [19].

The task graph can either be expressed implicitly, such as the Parameterized Task Graph (PTG) originally used for PaRSEC [20], while the majority use the STG model, as explained in the previous section, the STG approach provides a clear and familiar meaning to the program, which simplifies development and maintenance, as well as facilitates the porting of existing codes.

The major issue with the distributed STG approach is limited scalability for medium and fine-grained tasks. Taking into consideration the application nature and task granularity, OmpSs-2@Cluster, for example, the distributed-memory variant of OmpSs-2 [21], scales to about 16 nodes [19]. At the same time, other STF approaches that create tasks in parallel, such as OmpSs@cloudFPGA [22] and StarPU-MPI [11] achieve somewhat better scalability. However, they are still ultimately limited by the sequential filtering of task dependencies. Moreover, all nodes need to agree on the same mapping of tasks to nodes independently. This static or deterministic allocation of tasks to nodes makes it impossible to balance the load across nodes transparently.

A common HPC applications category is implementing iterative methods or multi-step simulations that create the same Directed Acyclic Graph (DAG) of tasks on each timestep. A recent work proposed the `taskiter` directive [1], which declares that a specific loop creates the same DAG of tasks and accesses on each iteration and that the program remains valid if the code inside the loop body but outside tasks is executed a single time. This information allows the runtime to execute the loop body once to create the tasks and dependencies for a single iteration. Then, the information that the subsequent iterations follow the same pattern is used to build a cyclic graph representation that describes the complete computation. This reduces the overheads of task creation, scheduling and dependency management by incurring these overheads only for the first iteration and amortizing their cost across all loop iterations.

Taskiter, despite originally designed for SMPs, is especially well-suited for distributed task-based approaches due to two key factors.

1. While the number of nodes grows, the number of tasks normally grows in proportion to occupy the computational resources, so the sequential bottleneck that motivates `taskiter` scales at least as fast as the number of nodes. Meanwhile, the wall–clock time for the computation either stays roughly constant (for weak scaling, i.e., fixed computation per node) or falls within the number of nodes (for strong scaling, i.e., fixed problem size). The result is that the growing sequential bottleneck quickly dominates the total execution time.

2. Knowing the full cyclic dependency graph in advance allows the runtime to precompute all MPI data transfers. However, the common approach is to compute data transfers dynamically while the graph is being built, resulting in a large number of control messages between the nodes.

Chapter 5 of the thesis extends the OmpSs-2@Cluster distributed tasking model to support a distributed form of taskiter and describes the full implementation. Our approach eliminates control messages to and from the master node, replacing them with peer-to-peer, non-blocking MPI calls that are transparently integrated into the application's task graph . By integrating the MPI communications directly into the application's task graph, our approach naturally overlaps computation and communication, in some cases exposing dramatically more parallelism than fork–join MPI + OpenMP.

1.3 Thesis Outline

This thesis is structured in six chapters, including this one, and it is organised as follows:

- Chapter 1 introduces, motivates and briefly describes this work in addition to outlining the general structure of this document.
- Chapter 2 provides an overview of fundamental concepts crucial for comprehending the core of this thesis, including the OmpSs-2 task-based programming model and OmpSs-2@Cluster distributed task-based models as well as Nanos6 and Nanos6@Cluster runtime systems implementation of the aforementioned tasking models respectively. In addition to the outline, the experimental hardware and software setup and benchmarks were used in this work.
- Chapter 3 is a review of the literature and state-of-the-art with emphasis on initiatives related to this thesis.
- Chapter 4 elaborates on the first contribution of this work of: the automatic data access aggregation via introducing the `auto` and `none` clauses as an extensions to the OmpSs-2@Cluster task-based programming model and implementing it inside the Nanos6@Cluster runtime references implementation of the OmpSs-2@Cluster model. This chapter is published in [23].
- Chapter 5 is the second contribution of this thesis, the `taskiter` directive for exploiting iterative applications. This chapter is published in [24] and is under review when writing this thesis.
- Chapter 6 concludes the thesis and suggests potential directions for future research.

1.4 Publication List

This thesis is based in part on the following papers:

① Omar Shaaban, Jimmy Aguilar Mena, Vicenç Beltran, Paul Carpenter, Eduard Ayguadé and Jesus Labarta Mancho. "Automatic aggregation of subtask accesses for nested OpenMP-style tasks". in IEEE 34th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD), 2022. Doi: 10.1109/SBAC-PAD55451.2022.00042.

This publication presents much of the material of Chapter 4 of the thesis. This material has also been reported in DEEP-SEA D5.6 (submitted to EC).

② Omar Shaaban, Juliette Fournis d'Albiat, Isabel Piedrahita, Vicenç Beltran, Xavi Martorell, Paul Carpenter, Eduard Ayguadé, and Jesus Labarta. "Leveraging Iterative Applications to Improve the Scalability of Task-Based Programming Models on Distributed Systems". Submitted with a major revision at the ACM Transactions on Architecture and Code Optimization (TACO).

This publication presents much of Chapter 5 of the thesis. This material has also been reported in DEEP-SEA D5.6 (submitted to EC).

There are two additional publications as second author

③ Jimmy Aguilar Mena, Omar Shaaban, Vicenç Beltran, Paul Carpenter, Eduard Ayguadé, and Jesus Labarta. "OmpSs-2@Cluster: Distributed memory execution of nested OpenMP-style tasks", in European Conference on Parallel Processing (Euro-Par), 2022.

This paper presents the basic design and implementation of OmpSs-2@Cluster. My contributions were in the design space exploration of the runtime with a contribution to the offloaded tasks by providing a more efficient offloading mechanism, which enabled the OmpSs-2@Cluster to use complex irregular data structures as dependencies for offloaded tasks. Part of this process involved the development of the DMRG, InfOli, smart mirror and n-body applications case studies for analysis, evaluation, and performance limitations of the proposed design. The InfOli application was reported in EuroEXA EU H2020 projectś Deliverable D2.6 [25] and the smart mirror application was reported in LEGaTO EU H2020 projectś Deliverable D3.4 [26].

④ Jimmy Aguilar Mena, Omar Shaaban, Victor Lopez, Marta Garcia, Paul Carpenter, Eduard Ayguadé, and Jesus Labarta. "Transparent load balancing of MPI programs using OmpSs-2@Cluster and DLB", in International Conference on Parallel Processing (ICPP), 2022.

This paper presents a solution for multi-node dynamic load balancing of MPI + OmpSs-2 programs using OmpSs-2@Cluster. This publication benefits from the analysis and performance improvements discussed above, and it uses the n-body benchmarks in the evaluation process.

CHAPTER 2

Background

This chapter describes the programming models, runtimes and libraries on which this PhD dissertation builds. It gives an overview of OmpSs-2 and OmpSs-2@Cluster, as well as the Nanos6 and Nanos6@Cluster runtime systems. It also describes Task-Aware MPI (TAMPI), a library that integrates tasking with MPI.

2.1 OmpSs-2 Parallel Programming Model

OmpSs-2 [21] is the second generation of the OmpSs [27] task-based parallel programming model developed by the Programming Models group of the Computer Sciences department at the Barcelona Supercomputing Center (BSC). OmpSs/OmpSs-2 descends from the design principles of two programming models: Star Superscalar (StarSs) and OpenMP [9], hence the name OpenMP Superscalar (OmpSs).

The main principles inherited from StarSs are tasking, dependencies and support for heterogeneity. These are integrated with OpenMP-style compiler directives (known as pragmas in C/C++), which extend the C/C++ and FORTRAN programming languages. Sequential code is annotated with pragmas to produce the same code's parallel version. A compiler that does not support OmpSs will ignore the annotations and generate a correct sequential program. This ease of use constitutes OmpSs's philosophy as a productive, high-performance programming model, with no need to redesign the code for a parallel version. Some of the features of OmpSs have been adopted by OpenMP, including tasking in OpenMP 3.0, task dependencies in OpenMP 4.0, and the `taskloop` construct in OpenMP 4.5. OmpSs-2 is considered an open-source research platform developed to explore and evaluate new features for potential future standardisation in OpenMP [28]. OmpSs-2 supports multiple parallel architectures, including multi-core SMPs, GPUs [4], FPGAs [6], and distributed-memory cluster architectures [7, 18].

2.1.1 Execution Model

OmpSs-2 uses a thread-pool execution model, which implicitly initiates parallel execution before the application starts executing. This can be contrasted with OpenMP's fork–join model, which explicitly outlines the beginning and end of each section of code to be parallelised.

Being a tasking-based model, OmpSs-2 expresses parallelism via tasks with dependencies between them. In OmpSs-2, all application code is contained in tasks, including the `main` function, which is wrapped inside a task. When the *main-task* executes, it calls the original `main` function. OmpSs-2 starts by initiating a pool of threads assuming that threads will divide into a single master thread and other threads acting as worker threads. The `main-task` is added to a queue for ready tasks that are picked up later by any of the worker threads, while the other worker threads will be waiting for other tasks to become ready for execution, either tasks created by the `main task`, or tasks created by the runtime for management activities. In OmpSs-2, synchronisation is implied by task data accesses; the task graph is created incrementally for each task created by checking its dependencies against the previously created tasks. Tasks are scheduled once all of their predecessors have been executed, and the `main-task` acts as the parent of all tasks in the application.

Similarly to OpenMP, OmpSs-2 defines particular tasking constructs, such as `taskloop` and `task for`. A `taskloop` construct divides the loop space into separate subtasks according to a grain size selected by the user. All subtasks will collaboratively execute a portion of the loop, and all subtasks will wait for other subtasks to finish executing their portion. A `task for`, which is a work-sharing task builder that can operate concurrently across multiple threads, much like OpenMP's parallel for. Unlike other tasks, `task for` tasks do not require all threads to work together and do not impose any barriers. The `task for` divides the iteration space of a for-loop into chunks that are then distributed among the collaborating threads.

OmpSs-2 also shares and defines common semantics of the OpenMP `taskwait` directive. In OmpSs-2 `taskwait`, suspend the current task execution until all of its children (prior to the `taskwait`) finish execution. However, in OmpSs-2, a `taskwait` does not execute at the end of the task's body; this allows the early finalisation of the task and the release of its dependencies.

2.1.2 Dependency Model

OmpSs-2 follows an asynchronous data-flow model between tasks. In OmpSs-2, task data accesses, which define data consumed by a task during execution time, are used as a single mechanism to compute dependencies that enforce task ordering, determine data locality, and identify data copies for tasks executing on multiple cluster nodes if required. Data dependencies are defined by the user in the task construct either by using `depend` clause as in Figure 2.1a or the OmpSs short notation as in Figure 2.1b. Both methods use a comma-separated list of the memory regions or can specify them individually.

```
depend(<type> : <memory-region-reference-list>)
```

(a) OmpSs-2 full notation syntax

```
<type>(<memory-region-reference-list>)
```

(b) OmpSs-2 short notation syntax

Figure 2.1 OmpSs-2 dependency annotations syntax comparison

The semantic of `depend` clause is extended with types `in`, `out`, or `inout`, that defines input (read), output (write), or input-output (read/write) data dependency over a user-specific data range. Dependency enforcement between a predecessor and successor tasks can be summarised according to the following rules:

- For an `in` dependence type, it imposes a dependency over a predecessor task that has previously defined either an `out` or `inout` dependence type on the same memory region.
- A task with an `out` dependence type enforces a dependency over a predecessor task with either `in`, `out`, or `inout` dependence type on the same memory region.
- The `inout` dependence type combines both types `in` and `out` semantics.

Moreover, OmpSs-2 defines additional types that add an extra degree of freedom to the dependency model, allowing for more parallelism in special cases:

- **`concurrent`**: The concurrent dependence type acts as a special **`inout`** which enforces dependencies between other types but not **concurrent** relaxing synchronisation between tasks; hence, it is the developer responsibility, in that case, to verify that the tasks can run concurrently; otherwise an extra synchronisation will be needed.
- **`commutative`**: The commutative dependence type functions akin to the **`inout`** type to **`in`**, **`out`**, and **`inout`** types. Additionally, it enforces a dependency over tasks that define a commutative type over the same memory regions. However, they can be executed in any order (the creation order is still preserved). Any permutation ordering of those tasks annotated with commutative is acceptable, provided that one task is executed at a time.
- **`reduction`**: is considered a **concurrent** type with special reduction operation.

The **`concurrent`**, **`commutative`**, and **`reduction`** types are also governed by the same rules for ordinary types regarding how to resolve the situation when a task has multiple accesses covering the same memory region, which we refer to as the *access upgrade rules* and shown in Table 2.1.

Table 2.1 Access upgrade rules, which define the combined effect of overlapping task accesses. This is the least restrictive access type that implies all ordering constraints and data transfers of the constituent accesses.

	in	**out**	**inout**	**concurrent**	**commutative**	**reduction**
in	in	inout	inout	inout	inout	invalid
out	-	out	inout	inout	inout	invalid
inout	-	-	inout	inout	inout	invalid
concurrent	-	-	-	conc.	comm.	invalid
commutative	-	-	-	-	comm.	invalid
reduction	-	-	-	-	-	reduction*

* Non-identical overlapping reductions on the same task are undefined.

Dependencies can expressed in two ways, either as *discrete*, or *fragmented-regions*, which unlike the *discrete* allow the memory regions overlapping. In both cases, dependencies can be defined either by start address only as in Figure 2.2a, or by precisely specifying the starting address and length of the memory regions accessed by the tasks as in Figure 2.2b, which is useful in case of cluster execution to identify the size of memory copies for offloaded tasks (see Section 2.3.2). All tasks in the application are linked with dependencies between them into a graph structure that we refer to as the *task dependency graph*. A task is considered ready and can start executing once all of its data dependencies are satisfied, meaning that no other predecessor tasks accessing this data and/or the data location are known in case tasks are offloaded to other devices or cluster nodes.

```
#pragma oss task <type>(<memory-region>[startAddress])
```

(a) Using starting address only.

```
#pragma oss task <type>(<memory-region>[startAddress:length])
```

(b) Using both starting address and length of the access.

Figure 2.2 Ways of specifying dependencies in OmpSs-2.

A common approach for parallelisation is to organise tasks into multiple levels; this involves breaking down high-level functions into smaller tasks, each representing a distinct unit of work. Subsequently, these tasks can be refined further by being decomposed into additional subtasks. OmpSs-2 extends OmpSs and OpenMP to improve task nesting and fine-grained dependencies across multiple nesting levels through the `weakin`, `weakinout` and `weakout` access types [10, 29], which indicate a task does not access the data, but its nested subtasks may do so. Any subtask directly accessing data needs to specify a dependency with a strong (non-weak) access type. Any task that delegates access to a subtask must include the data region in at least the weak variant. Weak accesses provide a link between the dependency domains at different nesting levels but do not delay parent task execution or require data transfers on distributed memory. Adding weak accesses exposes more parallelism, allows better scheduling decisions, and enables parallel instantiation of tasks with dependencies among them.

The taskwait directive in OmpSs-2 has a similar semantic and description to the one in OpenMP. Taskwait suspends the current task until all child tasks are generated before the taskwait is finalised. If the taskwait includes dependencies, the current task region is suspended until all the child tasks with dependencies on the declared dependency region are executed entirely.

As we mentioned, the weak parent tasks act as a link between subtasks in task nesting, meaning that any accesses that are directly accessed by the parent task which are not accessed by any of the subtasks can be released as the parent task finishes. At the same time, dependencies appearing in one subtask but not another can also be released immediately after that subtask finishes execution, allowing for more parallelism.

Consequently, it is recommended to avoid using taskwait at the end of the body of a parent task in nested tasks because taskwaits delay the finalisation of subtasks, hinder the discovery of parallelism and delay the release of all dependencies until all child tasks finish (even the dependencies not declared by any subtask).

2.2 OmpSs-2@Cluster

OmpSs-2@Cluster [7, 30] is the flavour of the OmpSs-2 parallel programming model that leverages multiple distributed memory cluster nodes. A program that has been written for OmpSs-2 to execute on SMP machine is compatible with OmpSs-2@Cluster. OmpSs-2@Cluster extends the OmpSs-2 with specific data-flow semantics to match the execution workflow of a cluster-based system. It can be considered as an alternative to explicit MPI + OmpSs-2 for small to medium size clusters [7, 24], depending on the algorithm and problem size. It supports active malleability, interacting with the job scheduler to request or release compute nodes and then making use of these resources in a way that is transparent to the programmer [19]. In addition, it can also be used to provide multi-node dynamic load balancing for MPI + OmpSs-2 programs [18], using its ability to offload tasks to ensure that each node has an equal amount of work.

The main OmpSs-2@Cluster features can be summarized as:

1. **Sequential semantics**:
   Sequential semantics refers to the behaviour of a system or program where operations are executed strictly sequentially, following the order specified by the program logic. OmpSs-2@Cluster program is simply the equivalent sequential program annotated with `#pragmas` directives, giving a clear and familiar meaning to the program, and it simplifies the porting of existing codes at any nesting levels.

2. **Common address space**:
   All nodes in an OmpSs-2@Cluster program see the same virtual address space at the same starting address, allowing data allocated on one node to be copied and accessed at the exact memory location on any other node without any address translation, and facilitates the use of complex-pointer data structures used by some irregular applications, and the porting of existing benchmarks into OmpSs-2@Cluster.

3. **Distributed memory allocation**:
   OmpSs-2@Cluster provide the user with a distributed memory allocation as an alternative allocator for larger data manipulated by multiple cluster nodes. This allocation scheme is distinctive in that there is no need to copy the data back since the distributed data is meant to be used locally on each node, especially in cases where a two-level nested task is used to distribute tasks over multiple nodes, which is the comm case encountered in this research. In addition, huge

allocations usually occur at the beginning of an application and at a non-frequent pace, making the centralised allocation model employed by OmpSs-2@Cluster efficient and with marginal overhead. Finally, it provides the user with allocation affinity hints to exploit specific commons distributions see Section 2.3.3.

4. **Minimize overall data transfers**:
The default behaviour of the `taskwait` synchronisation primitive used in a distributed memory environment may create undesired data transfers. OmpSs-2@Cluster extend the `taskwait` with two options to allow more control and flexibility:

   - **The `taskwait on`**, which copies back only a specific subset of the tasks's local data as specified to the `on` clause. The `on` syntax is similar to that used in any dependency declaration as in Figure 2.1.
   - **The `taskwait noflush`**, which will not perform any data copy.

5. **Control of dependencies release**:
Distributed memory systems rely on nested tasks for distributing work execution (Section 2.3.4); the parent task (first nesting level) must wait for all of the subtask (inner levels) to finish before it can pass these dependencies to successors in the task-graph. OmpSs-2@Cluster provide three levels that control the behaviour of subtasks dependencies release:

   - **Early release**, which releases sub-task dependencies to the parent successor as soon as the sub-task finishes execution, allowing the parent's successor to be ready earlier, even if there are other children subtasks still executing.
   - **Late release** is beneficial in cases where subtasks will end roughly simultaneously; hence, the early release will be an overhead rather than an advantage. OmpSs-2@Cluster achieves this via the `wait` clause that implies the parent task must wait for all of its children to finish before releasing their dependencies.
   - **Auto release**, is a hybrid of the early and late approaches. Offloaded tasks will have early release to successors on the same cluster node executing that offloaded task and late release for all other tasks.

6. **User exposed API and scheduler hints**: OmpSs-2@Cluster extends the existing API to provide the user with more options and system information that help the development process. In addition, the user is also provided with scheduling hints for choosing between a set of scheduling policies that can

help with situations which require specific control, such as with some irregular applications. In addition, the **`node(node-id)`** clause is provided to manually set the executing cluster node indicated by the `node-id`.

2.3 Runtime Reference Implementation

Nanos6 [31] is the runtime system that implements the OmpSs-2 task-based programming model for SMP architecture. Nanos6@Cluster extend Nanos6 runtime with the OmpSs-2@Cluster runtime specifications for distributed memory systems, and it is the OmpSs-2@Cluster reference implementation used in this work. Both Nanos6 and Nanos6@Cluster rely on the Mercurium [32] C/C++ and Fortran source-to-source compiler that transforms the high-level user code's directives (i.e. pragmas) into routine calls implemented by the Nanos6 and Nanos6@Cluster runtime systems. Nanos6, Nanos6@Cluster, and Mercurium are all developed by the Programming Models group of the Computer Sciences department at BSC.

2.3.1 Nanos6

The Nanos6 runtime library provides the essential infrastructure required to execute applications using OmpSs-2 tasks. This includes fundamental task creation, dependencies computation and registration, and linking different task dependencies into the task dependency graph. In addition, the runtime provides a scheduler that utilises the underlying hardware resources while respecting the execution order implied by the task dependency graph. It also contains other components to manage system resources, memory allocation, thread creation, and support for other devices and accelerators. In addition to being used as a runtime library, Nanos6 can also be used as an API for explicitly and manually handling task creation and submission to the scheduler. It offers different operation variants that can help the user at different development stages. This includes the debug variant, variants instrumentation modes such as Extrae [33], verbose, and linting [34], which can help to visualise the application performance by generating execution traces with different forms and levels of information. Moreover, it can be interfaced with external libraries such as DLB for load balancing [18]. Users can tweak different runtime configurations and options via a configuration TOML file.

A typical OmpSs-2 program flow will start by compiling the code using the Mercurium compiler that translates the user `#pragmas` into Nanos6 API routines as mentioned in section 2.3 in addition to wrapping the main function inside an implicit task (the main task 2.1.1) and change the starting address of the actual main to a `main-wrapper` function inside the runtime. The `main-wrapper` will bootstrap the runtime by allocating the required resources, initiating a pool of worker threads, and adding the `main-task` as a ready task to the ready tasks queue part of the runtime scheduler. One of the worker threads will pick up the `main-task` and start executing it immediately, which in turn will start the actual user main. The remaining worker threads will wait for other ready tasks that can be created by the `main-task` or other tasks created by main that are already currently running. Figure 2.3 shows an overview of OmpSs-2 application development process.

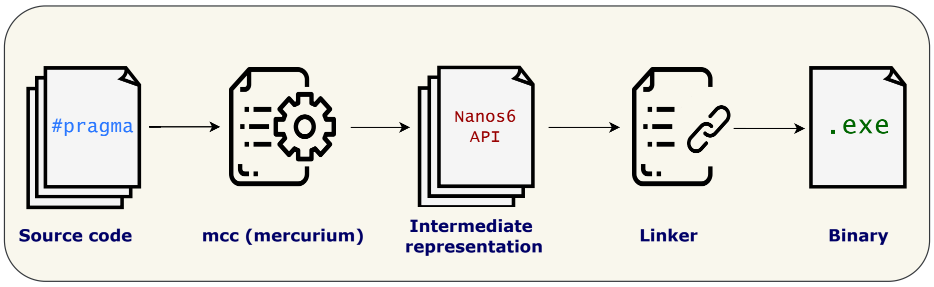


Figure 2.3 OmpSs-2 development flow

2.3.2 Nanos6@Cluster

The Nanos6@Cluster runtime executes tasks on multiple cluster nodes by providing a communication layer using an MPI library that is transparent to the user. The application will execute as a typical MPI + OmpSs-2 application without the need to manually write any MPI routines. In addition, a program written for Nanos6 can seamlessly run with Nanos6@Cluster with the only requirement to express the full dependencies as indicated by the *fragmented-regions* (section 2.1.2) as it will be used to calculate data copies required by tasks offloaded to/from other nodes in the cluster. The cluster mode of Nanos6 is enabled simply by an option in the configuration TOML file passed to a cluster-compatible build of Nanos6.

Nanos6@Cluster follows the same execution flow of that is in Nanos6 (Section 2.3.1) with the addition of that each cluster node will instantiate an instance of the runtime that will run by an MPI process as seen in Figure 2.4. The runtimes coordinate as peers, with all communication for control messages (see Section 2.3.4) and data transfers using two-sided MPI point-to-point communications. A dedicated thread on each cluster node handles the communications. All communication operations are transparent to the developer by the runtime, and the developer cannot directly use MPI routines in cluster mode. However, it is possible when the runtime uses the DLB library [18].

All nodes have the same copy of the runtime; only `Node 0` differs in that the node executes the `main-task` and manages extra activities such as collative synchronisation across all nodes required by the distributed memory allocator `nanos6_dmalloc` (Section 2.3.3) or the `nanos6_resize` routine required by malleability operations [19]. Tasks are scheduled locally to a node via the Nanos6 SMP scheduler or through the cluster scheduler for offloaded tasks.

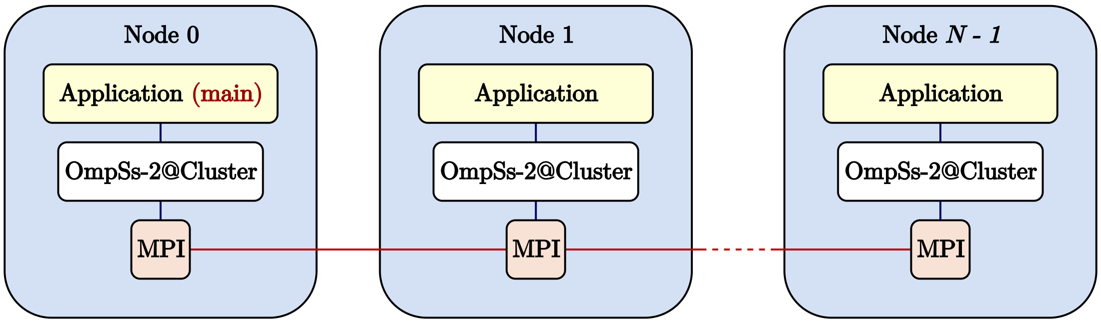


Figure 2.4 OmpSs-2@Cluster architecture in which each rank is a peer. The `main` function is executed as a task on Node 0. All other tasks may be offloaded for execution by any other rank.

2.3.3 Memory Model

As indicated in Section 2.2, Nanos6@Cluster employs a common address space on all nodes in the cluster, which starts at the same virtual memory address on all the nodes as well. The allocation is done via an `mmap` [35] call on a specific region that is collectively agreed upon on all the nodes. The memory allocation size and the starting address will have a default value. Otherwise, the user can manually set them in the TOML file. The common address space (which is a virtual memory) is partitioned into two types of memories, *Local Memory* and *Distributed Memory*. The *Local Memory*, and *Distributed Memory* are visible to the developer via `nanos6_lmalloc` and `nanos6_dmalloc` APIs respectively. Fixating the address space across all nodes allows data allocated on one node can be seamlessly accessed by tasks that execute on any other node [7]. This avoids address translation and allows direct use of existing data structures with pointers.

The *Local Memory* intended to be used by all computations within a single task scope and by input/output arguments of subtasks of the current parent task. As the name implies, the *Distributed Memory* is used for data distributed across all nodes. The *Local Memory* is further subdivided into regions as many as cluster nodes, and associate each region with each of the nodes as seen in the *Local Memory* layout Figure 2.5. The *Distributed Memory* also splits the distributed memory and assigns each node a region similar to the *Local Memory*. However, the partitioning of the allocated memory will be based on a user-defined *distributed policy* such as round-robin, block, or block-cyclic. Figure 2.6 shows a *round-robin* policy example of the distributed memory layout.

2.3.4 Tasks, Offloading and Scheduling

The `main` function executes as a task on the first process, Node 0. All other tasks are created as subtasks of their parent, initially on the node that executes the parent task. Top-level tasks are, therefore, initially created on Node 0. If the task is to be executed locally, it is passed to the host scheduler when it is ready, as usual. Otherwise, an MPI message is sent to the remote rank since the task will be executed remotely. On receipt, the remote rank creates a copy of the task, which is scheduled by the host scheduler on the remote rank. Scheduling is, therefore, done at two levels: the cluster scheduler of the parent's rank maps the task to the execution rank, and the host scheduler on the execution rank schedules the task to run on an available core.

Optimised OmpSs-2@Cluster programs typically have two levels of nested tasks. The top level has one offloadable task per process to distribute the work across processes, and the second level has a small number of non-offloadable tasks per core to distribute the work across the cores on that process. This approach mitigates the sequential task creation and offloading bottleneck at the cost of some programmer complexity, and it is responsible for a good part of the scalability of OmpSs-2@Cluster to about 16 nodes [7]. Since the top-level tasks do not themselves perform computation, they can

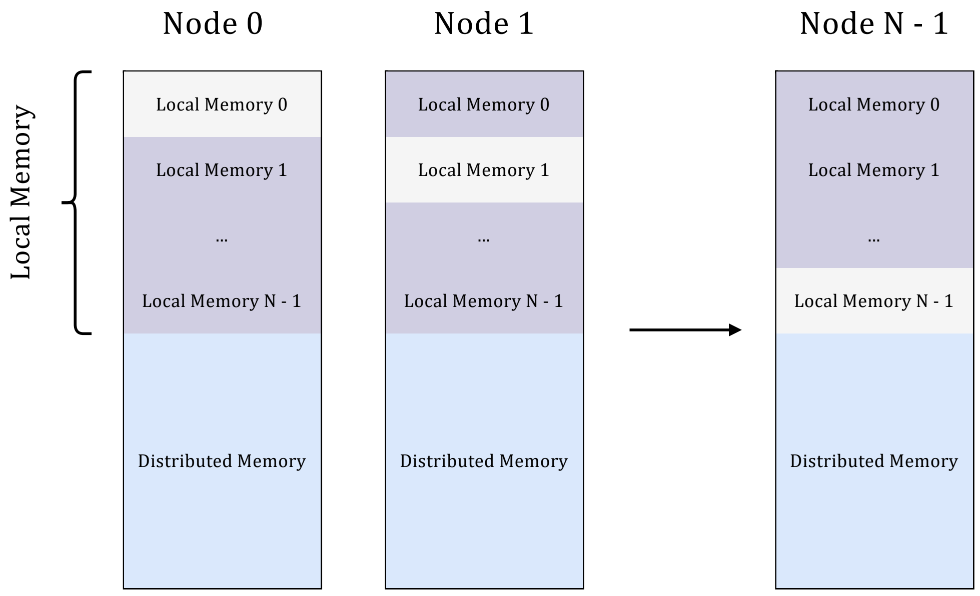


Figure 2.5 Nanos6@Cluster local memory model

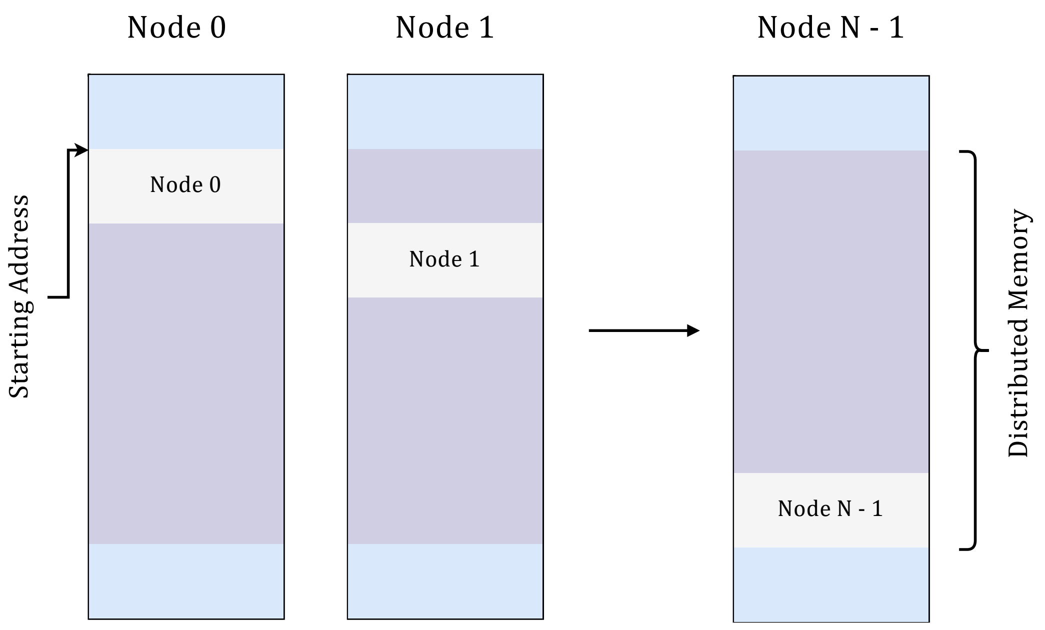


Figure 2.6 Nanos6@Cluster distributed memory model

execute and create their subtasks before the data is ready. This is made possible using weak accesses on the top-level tasks (see Section 2.1.2).

2.3.5 Control Messages

Control messages between nodes are used to offload tasks and maintain data dependencies between these tasks. Listing 2.1 is an OmpSs-2@Cluster program that creates two tasks, A and B, and offloads them from Node 0 to Node 1 and Node 2, respectively. These tasks have weak accesses, as discussed above in Section 2.3.4. The task execution rank is specified for concreteness in these examples by overruling the cluster scheduler using the `node` (see Section 2.2). Figure 2.7 shows the sequence of MPI messages, from top to bottom, involved in the execution. First, Task A and Task B are offloaded to Node 1 and Node 2, respectively, with two consecutive Task New messages. When Task A finishes on Node 1, a Task Finished message is sent back to Node 0. This message releases the output value of x and identifies the location of the latest version of the data. In turn, Node 0 sends a Satisfiability message to Node 2, which passes global write permission and the current location of x. Since the data is on Node 1, Node 2 sends a Data Fetch control message to Node 1, which responds by posting a point-to-point data transfer containing the data. Finally, when task B is complete, Node 2 sends a Task Finished message to Node 0. The runtimes collectively enforce a global ordering of writes, and they send and receive a total of seven messages, all except one (to offload B) on the critical path. Only one of these messages carries the actual data.

```
// Task A
#pragma oss task node(1) depend(weakout:x)
{ ... }

// Task B
#pragma oss task node(2) depend(weakin:x)
{ ... }
```

Listing 2.1 Source code of Figure 2.7

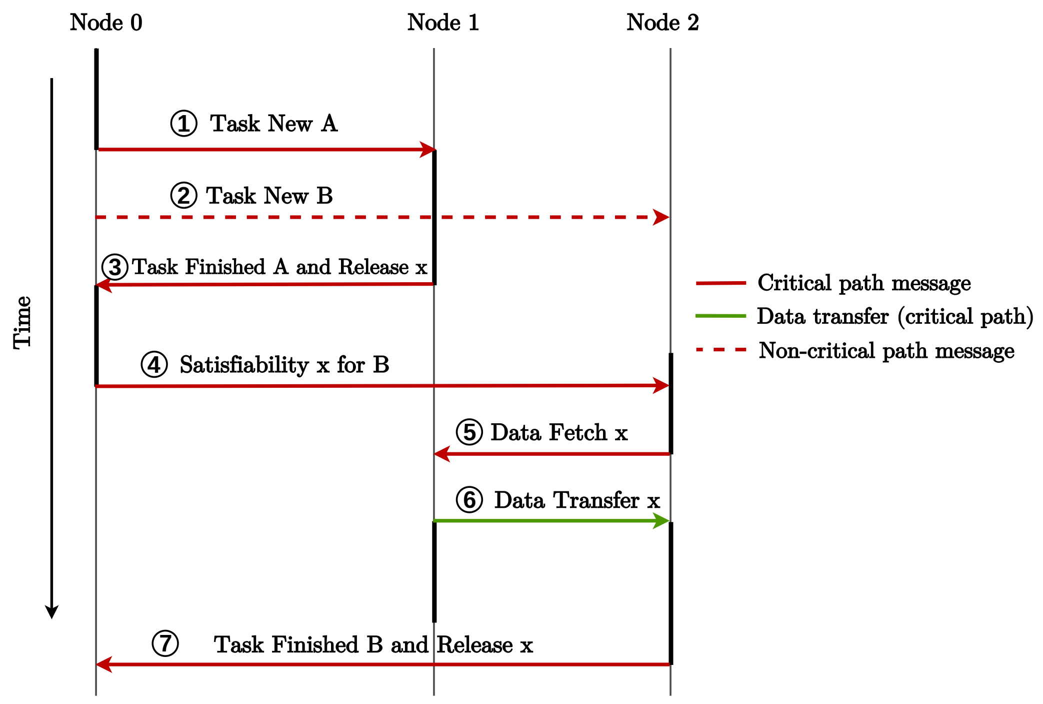


Figure 2.7 Large number of MPI messages for OmpSs-2@Cluster. Offloading and executing these two tasks requires one data transfer message and six control messages.

2.4 Taskiter

The `taskiter` is an iterative construct that was recently proposed [1] for OmpSs-2 and OpenMP. The semantics of the `taskiter` indicate that each iteration of the associated loop creates the same dependency graph of tasks and accesses which can be created once and re-used for subsequent iterations. Figure 2.8a and 2.8b show the syntax, for OpenMP and OmpSs-2 respectively. The *loop* can be any loop statement, so long as (a) each iteration of the associated loop creates the same dependency graph of tasks and accesses at the top level, and (b) the program remains valid if the code inside the loop body but outside any task is executed just once. Condition (a) restricts only the top-level dependency graph: top-level tasks may create nested subtasks with different dependency graphs in different iterations. In addition, the `taskiter` can be combined with the optional `unroll(n)` clause, which enables `taskiter` to support a loop whose task graph repeats every n iterations.

```
#pragma omp taskiter [clause [...]] new-line
    loop
```

(a) Taskiter clause (OpenMP)

```
#pragma oss taskiter [clause [...]] new-line
    loop
```

(b) Taskiter clause (OmpSs-2)

Figure 2.8 OpenMP and OmpSs-2 taskiter construct syntax

An example program using taskiter is shown in Figure 2.9a, where the only modification to take advantage of taskiter is the pragma annotation on line 1. Figure 2.9b shows the regular task graph. When using taskiter, however, the runtime only executes one iteration of the taskiter's loop, creating the dependency graph for a single iteration. It then converts the graph into the Directed Cyclic Task Graph (DCTG) shown in Figure 2.9c.[1] In this figure, the non-cyclic edges between tasks in the same iteration are shown as solid lines, and the cyclic edges, passing from one iteration to the next, are shown as dashed curves.

By only creating tasks and computing dependencies for a single iteration, taskiter significantly reduces the sequential overhead. Moreover, since the DCTG is constant for all iterations, it is stored simply, without locking or complex lock-free data structures.

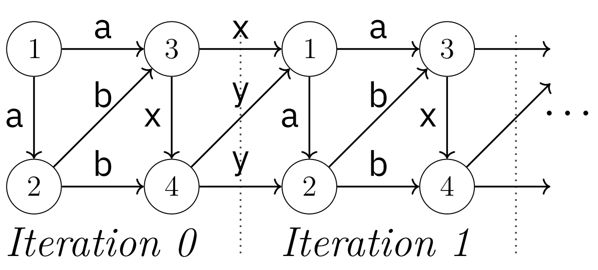
[1]The graph created by the runtime may have additional edges for the Write-after-Write (WaW) dependencies between consecutive iterations of the same task. These WaW dependencies are implied by existing paths in the graph and have been omitted from both subfigures.

This also reduces the impact on the execution time of the more powerful but expensive fragmented regions dependency system [36] since dependency system operations are only performed for the tasks in a single iteration.

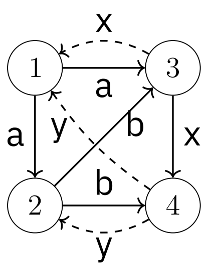
As normal, the tasks can create subtasks. Although the first-level task graph must be the same for each iteration, their subtasks, if any, may differ in each iteration, allowing irregularity between iterations at deeper nesting levels.

```
 1  #pragma omp taskiter
 2  for(int it=0; it < NUM_ITERATIONS; it++)
 3  {
 4      // Task 1
 5      #pragma omp task depend(in:x,y) depend(out:a)
 6      { ... }
 7
 8      // Task 2
 9      #pragma omp task depend(in:a,y) depend(out:b)
10      { ... }
11
12      // Task 3
13      #pragma omp task depend(in:a,b) depend(out:x)
14      { ... }
15
16      // Task 4
17      #pragma omp task depend(in:x,b) depend(out:y)
18      { ... }
19  }
```

(a) Example OpenMP program using taskiter construct



(b) Normal task dependence graph without taskiter



(c) DCTG with taskiter

Figure 2.9 Example OpenMP program using taskiter. The taskiter annotation on line 1 of subfigure (a) enables the runtime to replace the normal unrolled task graph in subfigure (b) with the concise directed cyclic task graph (DCTG) illustrated in subfigure (c).

Taskiter does not require a constant runtime number of iterations. If the compiler cannot determine the number of iterations, then the compiler inserts a special task known as a control task. The control task depends on every subtask in the current iteration as well as the control task from the previous iteration. The body of the control task checks the loop's condition and it cancels the taskiter when the condition is false. When the taskiter has the `unroll` clause, these control tasks are strided by the unrolling factor, providing a means to overlap tasks from different iterations.

2.5 Task-aware MPI (TAMPI)

Hybrid MPI+X applications are typically structured as alternating fork–join computation and sequential communication phases. This incurs additional synchronization, which, as we confirm in our results, hinders inter- and intra-node parallelism. We evaluate our approach primarily in comparison with fork–join MPI+OpenMP, but since OmpSs-2@Cluster is naturally asynchronous, we also compare with state-of-the-art asynchronous TAMPI.

TAMPI [15] is a library that integrates blocking and non-blocking MPI primitives with task-based programming models. It introduces a new level of MPI threading support, known as MPI_TASK_MULTIPLE. An application that requests this threading level can safely use blocking MPI primitives inside tasks without the risk of deadlock. Without TAMPI, a blocking MPI primitive blocks the task and the underlying thread that runs it. Even if a normal MPI implementation avoids busy waiting, allowing the hardware thread to become idle, the task-based runtime cannot discover that the hardware thread is available. TAMPI uses the MPI Profiling Interface (PMPI) interface to intercept MPI calls, and it releases any blocking thread to the runtime system to execute other tasks.

TAMPI also simplifies and optimizes the use of non-blocking MPI primitives by making their completion visible to the dependency system. This is done using the new TAMPI_Iwait and TAMPI_Iwaitall calls, as illustrated in Listing 2.2. The task on line 5 posts the non-blocking MPI_Irecv on line 8 to receive the contents of an array. It then calls TAMPI_Iwait, on line 9, which informs TAMPI that the given MPI request is associated with a dependency to the subsequent task (it comprises the output dependency on x). TAMPI uses the Nanos6 external events API [16] to delay the release of the current task's dependencies. The call to TAMPI_Iwait is non-blocking, so the task continues, finishing immediately and freeing its data structures and stack. Later, when the MPI request is complete, it is unnecessary to unblock and re-schedule the first task. TAMPI will use the external events API to release its dependencies; at this point, the task on line 12 can begin execution.

```
 1      double x[10];
 2
 3      ...
 4
 5      #pragma omp task depend(out:x)
 6      {
 7              MPI_Request request;
 8              MPI_Irecv(&x, 10, MPI_DOUBLE, other_rank, tag,
                    MPI_COMM_WORLD, &request);
 9              TAMPI_Iwait(&request, MPI_STATUS_IGNORE);
10      }
11
12      #pragma omp task depend(in:x)
13      {
14          ...
15      }
```

Listing 2.2 Example OpenMP program using Task-aware MPI's (TAMPI's) non-blocking communication. The call to TAMPI_Iwait makes the completion of the MPI_Irecv call on line 8 visible to the runtime's dependency system, simplifying the code and reducing task scheduling costs.

CHAPTER 3

Related Work

As OpenMP gains popularity in High Performance Computing HPC, task-based programming models have emerged as versatile, asynchronous approaches, providing productivity and performance advantages for SMP systems, often combined with MPI for distributed-tasking across multi-node clusters, prompting investigations into distributed-tasking's optimisation, runtime factors, and static parallelisation methods.

In this chapter, we explore the literature for related work and state-of-the-art solutions of distributed-tasking approaches and ways of dependencies specifications.

3.1 MPI, PGAS and Hybrid Approaches

MPI [37, 38] is by far the most widely used standard for writing HPC applications [39, 40], and it is well supported on all HPC systems. It is based on a distributed memory model with processes communicating via messages. Partitioned Global Address Space (PGAS) languages [41–44] and libraries [45, 46] provides a global address space so that the processes access remote data directly, through language constructs or an API, rather than communicating via messages. This requires a more advanced understanding of memory consistency and synchronisation. Both approaches, MPI and PGAS, place a high burden of data distribution, synchronisation and load balancing on the programmer.

"MPI + X" models, which combine MPI with shared memory parallelism via OpenMP [2], OpenACC [47], Compute Unified Device Architecture (CUDA) [48], or similar, have been under study for at least twenty years [49, 50]. Many applications use a fork–join approach [51, 52], where processes alternate between sequential communication and parallel computation phases, which hinders inter- and intra-node parallelism.

Habanero-C MPI (HCMPI) [53] automatically supports fine-grained overlapping of communication and computation, as it converts each MPI call into an asynchronous task. TAMPI [15] (see Section 2.5) is a more flexible approach that allows tasks to safely and efficiently call MPI primitives. All MPI+X approaches suffer from the fundamental issues of MPI, i.e., the programmer has to handle data distribution, synchronisation and load balancing, and inserting message sends and receives. They also require the programmer to split the parallelism between shared and distributed memory models.

DASH [54] provides a C++ template library for distributed memory, which is based on tasks in a PGAS model. Each process concurrently creates its task dependency graph. Tasks primarily access local memory, but they can also depend on memory owned by another rank. Execution is divided into phases, and dependencies between tasks in different processes are only resolved at the boundaries between phases. Each rank communicates its non-local dependencies to the rank that owns the data, and the data values are exchanged as soon as they are available. This model supports tasks in a global memory space, but the programmer is still responsible for data distribution and load balancing. It is also necessary to divide the program into phases, during which there is no inter-node communication.

3.2 Distributed Tasking

Distributed tasking approaches execute tasks with dependencies in a single task graph, which provides unambiguous dataflow semantics among all tasks of all processes. The task graph may exist only implicitly, based on a model of the structure of typical programs, but more commonly, the model uses a Sequential Task Graph (STG) formulation. In the latter case, the STG is constructed sequentially at runtime based on annotations or API calls. This may happen concurrently on all processes, creating a duplicate task graph in each process, or the task graph may be created by a single process, which distributes work to the other processes. The whole unrolled graph for an STG program is built task-by-task. However, it typically never exists in its completed form since tasks are added (constructed) and removed (after execution) concurrently.

3.2.1 Implicit task graph creation

PaRSEC [55, 12] is a distributed task-based model designed for scalability on distributed heterogeneous architectures. It is the basis for the DPLASMA library for dense linear algebra, which was the original motivation for PaRSEC. Its original formulation builds a parameterised DAG [20], which algebraically describes the dependencies between tasks in terms of the iteration variables. This model makes it very complex to develop programs, and it is not as expressive as other task models, being tailored for affine loops that are amenable to polyhedral analysis [56]. Our approach also avoids unrolling the whole dependency graph, but it is simpler and specialised for iterative applications, and it requires just one pragma to identify such a loop.

3.2.2 Concurrent and duplicated task graph creation

Several approaches build the same task graph, containing all top-level tasks and dependencies concurrently on all processes. All processes independently determine the same deterministic mapping of tasks to rank and execute only the tasks mapped to the current rank. Processes also insert appropriate send and receive primitives to pass data to and from tasks executed by other ranks.

StarPU-MPI [11] extends StarPU [57] to support distributed memory tasking using MPI. In this framework, top-level tasks are mapped to nodes using an owner-computes model. **YarKhan**'s [58] extension of QUARK [59] (a runtime environment for dynamic scheduling) uses a deterministic mapping of tasks to rank based on data distribution, and it also uses MPI for communication. **TBLAS** [60] takes a similar approach to target clusters of CPUs, each with multiple GPUs. **OmpSs@cloudFPGA** [22] targets clusters of FPGAS with direct FPGA-to-FPGA communication and hardware acceleration to mitigate the cost of filtering task accesses. **DuctTeip** builds hierarchical data structures and task graphs, mitigating the sequential bottleneck through a task nesting approach. **PaRSEC** [55, 12] also supports Dynamic Task Discovery (DTD) as an alternative to the Parameterized Task Graph (DTD). DTD constructs a general task graph from a sequential program, unrolling the full task dependency graph on each process. This is similar to the previously described approaches and suffers from the same bottleneck and flexibility issues.

In all the above approaches, all ranks must independently determine the same mapping of a task to rank, which makes it impossible to load balance dynamically. Every rank has to check the dependencies of every top-level task in every iteration, which limits the scalability for fine- and medium-grained tasks. Our approach has the advantage that it does not require task nesting with the possibility to support it (if needed), it is compatible with partitioning and re-partitioning of the cyclic task graph, and the entire overhead to build and manage the task graph and insert communication is amortised across all loop iterations. Finally, it interoperates with the fully general OmpSs-2@Cluster approach, which starts from a single sequential thread.

3.2.3 Sequential task graph creation

Other approaches build the top-level task graph on a single node, with task offloading to other nodes. **OMPC** [13] extends the LLVM OpenMP implementation of the target library with new target for offloading tasks and introduces the concept of a "remote device" for offloading tasks to remote nodes.

OmpSs-1@Cluster [61] and its successor **OmpSs-2@Cluster** (Section 2.2) extend BSC's OmpSs programming model to support distributed memory clusters. Both create the dependency graph on a single core, although OmpSs-2@Cluster has improved support for task nesting as a way to parallelise the creation of tasks on multiple ranks to reduce the pressure on the first rank. Our approach is compatible with OmpSs-2@Cluster, but it avoids all of the control message overhead inside the timesteps of iterative applications (see Section 2.3.5). We compare our results against OmpSs-2@Cluster in detail in Section 4.6, and demonstrate that while the existing OmpSs-2@Cluster approach is a viable alternative to MPI+OpenMP on up to 4 or 8 nodes, our approach is close to MPI + OpenMP on up to at least 32 nodes.

Legion [62] is a framework for parallel tasking computations on distributed heterogeneous systems. Execution begins on a single rank, and tasks are offloaded to other ranks. It supports task nesting and adopts a data-centric approach, where developers describe the structure and properties of data so that the scheduler can optimise data locality. Programs can either use Legion's native C++ API or the high-productivity **Regent** language [63]. It has disadvantages similar to OmpSs-2@Cluster in terms of scalability and the need to use task nesting to get good performance.

3.3 Other Approaches

3.3.1 Frameworks and Libraries

Charm++ [64] is an asynchronous execution model for HPC, based on migratable objects known as "chares". Chares communicate by exchanging messages, resulting in a form of concurrent and asynchronous execution that has some similarities to task execution without dependencies. **HPX** [65] is a C++ library that supports parallel computations using an interface that aims to be compatible with the C++ Standard Template Library (STL). It adopts an asynchronous and distributed task-based model that is expressed using futures and which supports data dependencies among futures. **X10** [66] is an object-oriented programming language for high-productivity programming that spawns asynchronous computations, with the programmer responsible for

PGAS data distribution. **Chapel** [67, 68] is a high-productivity HPC programming language, which allows asynchronous tasks to be spawned and executed on other distributed-memory nodes. Chapel's *sync* qualifier can block tasks until the necessary data is ready, allowing for coordinated execution order among tasks. Unlike other tasking models, a task graph is not created in advance of task execution.

Other include The Intel Threading Building Blocks (**oneTBB**) [69], Microsoft Task Programming Library (TPL) [70] and Google TensorFlow [71]. The TBB, TPL and TensorFlow share the goal of optimising parallelism and concurrency in computing tasks. TBB (for C++ developers) and TPL (for C# developers) are primarily focused on facilitating parallel execution of tasks by abstracting away low-level threading details, providing constructs like tasks, parallel loops, and dataflow parallelism. Similarly, TensorFlow, though primarily known for its role in machine learning, also emphasises parallelism through its computational graph abstraction and execution engine, enabling efficient execution of operations across multiple devices like CPUs and GPUs. All three frameworks aim to enhance performance and scalability by leveraging parallelism and concurrency, albeit in different domains and contexts.

3.3.2 Scripting and Workflows

Many scripting and workflow frameworks also adopt a distributed tasking approach with a directed acyclic graph of tasks and dependencies. **COMPSs** [72] is a Java, C/C++ and Python framework to run parallel applications on clusters, clouds and containerised platforms. It is a sequential task-based model similar to OmpSs, but dependencies are tracked through files or objects rather than the program's virtual address space. **Pegasus** [73] is another workflow management system that uses a DAG of tasks and dependencies. **GPI-Space** [74] is a fault-tolerant execution platform for data-intensive applications. It supports coarse-grained tasks that help decouple the domain user from the parallel execution of the problem. In all these approaches, the task granularity is much coarser than that targeted by our approach, with individual tasks lasting up to hours or days. It is viable for a single master node to manage all task scheduling and data transfers.

3.4 Data Access Specifications

Data access specification refers to the ways of providing data dependencies information that implies and ensures correct execution order between computing entities, such as tasks in the case of the OpenMP and OmpSs-2 models. This information can be provided at compile-time, which relies on static analysis of the source code, at run-time, discovers dependencies during the program execution, or via specific analysis tools and instrumentation, which we refer to as automatic.

3.4.1 Compile-time

Many research efforts target automatic compile-time parallelisation, e.g., Cetus [75], DawnCC [76, 77], AutoPar [78], Pluto [79] and TaskMiner [80]. These primarily target data parallelism, and only TaskMiner is specific to tasks. While some overlap with our work, the purpose is very different. These tools target the conversion of sequential to parallel code. As they analyse the code at compile time, there is no runtime overhead, but they typically only support code that follows a particular structure. Our `auto` clause does not determine the strong accesses of the subtasks that do the work. Instead, it aggregates information already known to the runtime so that fine-grained dependencies can be determined among subtasks that different concurrent parent tasks have created.

3.4.2 Run-time

OpenMP 5.1 [9] introduces `omp_all_memory`, which matches all accesses of previous sibling tasks. It is a convenient way to enforce a dependency that serialises with all prior tasks with specified access. Our proposal does not imply any ordering with respect to previous tasks. Instead, it is a way to indicate that some of the weak accesses on behalf of subtasks have not been specified. We are aware that no related work has solved the same problem.

3.4.3 Automatic

Tareador [81] uses an LLVM compiler stage to instrument all read–write instructions in a sequential application. This instrumentation can determine the strong data accesses and help explore potential parallelism strategies. It has been successfully used for this purpose in undergraduate courses on parallelism. StarSsCheck [82] is a tool based on Valgrind [83] that verifies the correctness of the strong accesses in a task-based program.

Linter [84, 34] is a runtime dynamic binary instrumentation tool that addresses the same problem. These tools all introduce enormous performance overhead of at least an order of magnitude. Our results demonstrate that our approach has a dramatically smaller overhead by concentrating on determining the weak accesses and assuming that the strong accesses are correct.

CHAPTER 4

Automatic aggregation of data accesses

4.1 Introduction

Maximising the underlying hardware utilisation requires creating enough work to keep all processing elements busy (e.g. CPU cores). This is a synonym for creating enough tasks to keep all the cores busy in task-based programming models. A solution is to use nested tasks to build a full dependency graph in parallel by concurrently executing multiple parent tasks. Tasking models such as OpenMP and OmpSs-2, however, restrict task creation until the addresses and sizes of all its accesses are known. This includes the parent task accesses, and all the accesses of its subtasks, their subtasks, and so on, also need to be inside specified regions. This restriction only applies to the dependencies of the created tasks and not all accesses in the program.

This chapter introduces the `auto` and `none` clause, addressing the aforementioned limitations that have been introduced in more detail in Chapter 1.

4.2 Motivation

4.2.1 Precise specification of data accesses without taskwaits

Listing 4.1–4.3 shows three versions of an example program that performs a matrix multiplication to calculate the upper triangle of symmetric $C = AB$ followed by a Cholesky decomposition of C. The three matrices are stored as hypermatrices, i.e., element `A[i][j]`, `B[i][j]` or `C[i][j]` is either `NULL` or a pointer to the actual block. For simplicity, only the data accesses involving matrix C are shown. We identify tasks using the OmpSs-2 `label` clause, which provides a string literal that can be used by a performance or debugger tool to identify the tasks in a human-readable format. In the spirit of building the dependency graph in parallel, the tasks in each row of matrix C are created by a different parent task, labelled `row`.

Listing 4.1 has a precise specification of the data accesses of the `matmul` and `potrf` tasks, but it requires the taskwait on line 21. There is otherwise no way to connect the dependency on the actual blocks, written `*C[i][j]`, from `matmul` to `potrf`. The taskwait is needed to ensure that the main program creates the `potrf` tasks with the `*C[i][j]` access at the correct address allocated on line 9, and that these tasks cannot be executed until the block has been written. The accesses to `*C[i][j]` are still required to serialise the `matmul` tasks that contribute to the same block of C and to manage the dependencies among the tasks performing the Cholesky decomposition (only `potrf` is shown).

```
1  typedef double Block[TS][TS];
2  Block *A[NT][NT], *B[NT][NT], *C[NT][NT];
3  // Matrix multiplication
4  for (int i = 0; i < NT; i++)
5  {
6      #pragma oss task depend(out : C[i][0; NT]) label("row")
7      for (int j = i; j < NT; j++)
8      {
9          C[i][j] = calloc(1, sizeof(Block));
10         for (int k = 0; k < NT; k++)
11         {
12             if (A[i][k] && B[k][j])
13             {
14                 #pragma oss task depend(inout : *C[i][j]) label("matmul")
15                 dgemm(A[i][k], B[k][j], C[i][j]);
16             }
17         }
18     }
19 }
20
21 #pragma oss taskwait
22
23 // Cholesky decomposition
24 for (int k = 0; k < NT; k++)
25 {
26     if (C[k][k])
27     {
28         #pragma oss task depend(inout : *C[k][k]) label("potrf")
29         potrf(C[k][k]);
30     }
31     // Rest of Cholesky
32 }
```

Listing 4.1 Approach with additional synchronisation (taskwait)

Despite being able to connect the dependencies from `matmul` to `potrf`, the `taskwait` impedes the concurrent creation and execution of tasks, and parallelism drops to zero waiting and is limited by the slowest task before the `taskwait`. Figure 4.1 shows an execution trace of the Listing 4.1 and how all the `matmul` must all finish before the `potrf` can start executing after the `taskwait`.

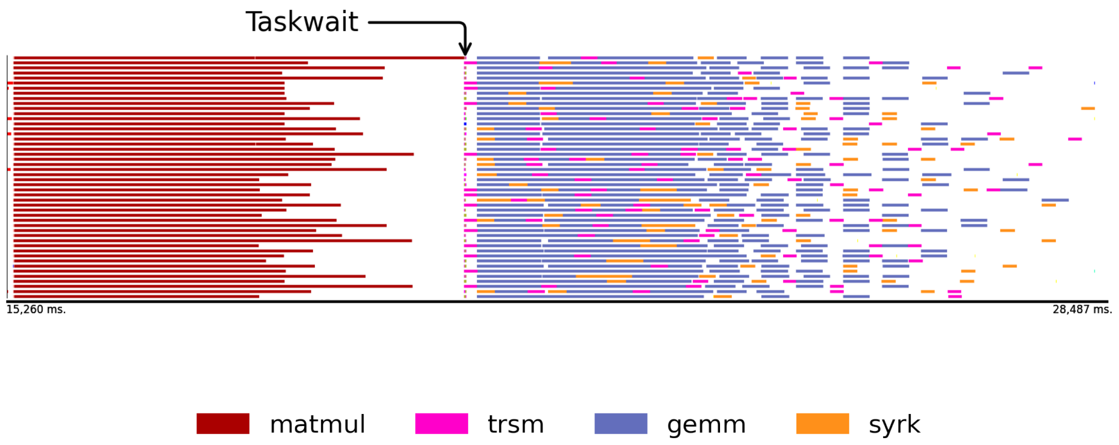


Figure 4.1 Extrae/Paraver trace of example in Listing 4.1. The execution time of `potrf` is very short, so these tasks are not visible in the traces.

A workaround might be nesting `potrf` inside a parent task that has a strong access on `C[i][j]`. However, the taskwait cannot be avoided in this case either. While this would correctly delay the creation of `potrf` and materialise its access to `*C[i][j]` at the right address, it will not enforce any dependencies between `matmul` and `potrf`. This is because these dependencies can only be linked if both their parents, `row` and the parent of `potrf`, have data accesses to `*C[i][j]` in at least a weak variant. We have simply moved the original problem, the data access to `*C[i][j]` at an unknown address, from `potrf` to its parent. Adding the `wait` clause to `row`, which disables early release, would work, but it would delay the execution of `potrf` until a whole row of C has been written. Moreover, it is complex even for this simple example; the dependence is easy to miss, and the resulting code would be obscure and error-prone.

Listing 4.2 shows a conventional solution using a sentinel. We replace each access on the block `*C[i][j]` with an access on the pointer to the block, `C[i][j]`. Tasks `matmul` and `potrf` have `inout` accesses on the pointer `C[i][j]`, not because they modify the pointer but because they modify the block that it points to. Although sentinels are commonly used this way, this approach has three problems. First, it breaks the idea that data accesses are a unified method to specify ordering constraints, data affinity and data transfers. Since the data accesses are "fake" and only for correctly enforcing ordering constraints, the runtime cannot optimise data locality properly. In particular, the runtime cannot take account of Non Uniform Memory Access (NUMA) affinity on the block, `*C[i][j]` when scheduling `matmul` and `potrf`. The program also becomes unsuitable for task offloading via OmpSs-2@Cluster. Secondly, the direct connection between the pragma annotations and the behaviour of the task is broken, reducing clarity. Thirdly, if the `matmul` and/or `potrf` tasks were decomposed into smaller subtasks, the use of sentinels would have to be redesigned throughout the whole program to enable fine-grained dependencies among these subtasks.

```
// Matrix multiplication
for (int i = 0; i < NT; i++)
{
    #pragma oss task depend(out : C[i][0; NT]) label("row")
    for (int j = i; j < NT; j++)
    {
        C[i][j] = calloc(1, sizeof(Block));
        for (int k = 0; k < NT; k++)
        {
            if (A[i][k] && B[k][j])
            {
                #pragma oss task depend(inout : C[i][j]) label("matmul")
                dgemm(A[i][k], B[k][j], C[i][j]);
            }
        }
    }
}
// Cholesky decomposition
for (int k = 0; k < NT; k++)
{
    #pragma oss task depend(inout : C[k][k]) label("potrf")
    if (C[k][k])
    {
        potrf(C[k][k]);
    }
    // Rest of Cholesky
}
```

Listing 4.2 Approach with a "fake dependency" (sentinel)

Finally, Listing 4.3 shows our proposed solution using `auto`. The precise semantics of `auto` will be described in Section 4.3. The `row` tasks have an output access on the pointers, `C[i][0;NT]`, allocated by the task, as well as an `auto` access to cover all the data accesses of their `matmul` subtasks, which are unknown at the time that the `row` tasks are created. The `none` access is an optimisation to enable the `row` tasks to run concurrently, and it is described below. The `row` tasks create the `matmul` subtasks that will perform the matrix multiplication, and they finish without waiting for these subtasks to complete. Since the first `weakpotrf` task has a strong access on the pointer `C[0][0]`, it can execute and create its subtask as soon as the first `row` task finishes. When all the `matmul` tasks that calculate this block are complete, the first `potrf` task can begin execution due to the dependency on `*C[0][0]`.

```
// Matrix multiplication
for(int i=0; i<NT; i++) {
    #pragma oss task depend(out: C[i][0;NT]) depend(auto) depend(none: C[0;NT][0;NT])
         label("row")
    for(int j=i; j<NT; j++) {
        C[i][j] = calloc(1, sizeof(Block));
        for(int k=0; k<NT; k++) {
            if (A[i][k] && B[k][j]) {
                #pragma oss task depend(inout: *C[i][j]) label("matmul")
                dgemm(A[i][k], B[k][j], C[i][j]);
            }
        }
    }
}
// Cholesky decomposition
for (int k=0; k<NT; k++) {
    #pragma oss task depend(in: C[k][k]) depend(auto) label("weakpotrf")
    if (C[k][k]) {
        #pragma oss task depend(inout: *C[k][k]) label("potrf")
        potrf(C[k][k]);
    }
    // Rest of Cholesky
}
```

Listing 4.3 Approach with proposed `auto` data accesses

The `none` access (see Section 4.3.2) on `row` is an optimisation to allow concurrent creation of the full task dependency graph. It does not affect the fine-grained dependencies among the `matmul`, `potrf` and similar tasks (not shown) that do the majority of the work. A `none` access indicates that no accesses need to be inferred in the region beyond those explicitly expressed by the other task accesses. In particular, the `row` tasks do not create any subtasks that access any elements of C, except possibly `C[i][j]`. Without `none`, the runtime would have to consider this possibility, in which case the sequential ordering rules would require an ordering dependency between a subtask of one `row` task and a later `row` task. Since this situation remains a possibility until the earlier `row` task is completed, the overall effect would be to serialise all the `row` tasks. The `none` clause says that this cannot happen, so all the `row` tasks can be executed concurrently to build the dependency graph in parallel.

The version using the `auto` clause has several advantages. Firstly, the data accesses unify the information specification needed to enforce task ordering, program data transfers, and optimise data locality. Secondly, the annotations are clear and flexible because they match the task's actual accesses. Thirdly, the task dependency graph can be constructed in parallel and well before task execution, maintaining parallelism and providing maximum ability for the scheduler to optimise load balance and data locality.

4.2.2 Productivity and Incremental Path for Nested Tasks

Listing 4.4 shows an example program with nested tasks. Task `parent` creates several tasks, each with label `child`, to do some of its work. In order to be properly nested according to the OmpSs-2 nesting rules, `parent` must itself have accesses, in at least the weak variant, covering all the accesses of its subtasks. This requires the multidependency on line 1, which burdens the programmer. It is redundant, as it duplicates information that the runtime can discover, and it is time-consuming and error-prone to write these annotations for all the parent tasks.

```
 1 #pragma oss task depend(weakin : {a[i][0; len[i]], i = 0; N}) label("parent")
 2 {
 3     // ...
 4     for (int j = 0; j < N; j++)
 5     {
 6         #pragma oss task depend(inout : a[j][len[j]]) label("child")
 7         {
 8             // Update a[j][0] ... a[j][len[j]]
 9             // ...
10         }
11     }
12 }
```

Listing 4.4 Parent task requires a multi-dependency for its children

Listing 4.5 shows the same example using the proposed `auto` dependency clause. It is clear that this approach allows a functional version to be obtained with much less effort. In terms of the ordering of the subtasks that do the majority of work, as well as data locality and data transfers, the behaviour is the same as that of Listing 4.4. The only cost is a small amount of overhead, which may, if necessary, be incrementally reduced by refining the task accesses guided by performance analysis.

```c
#pragma oss task depend(auto) label("parent")
{
    // ...
    for (int j = 0; j < N; j++)
    {
        #pragma oss task depend(inout : a[j][len[j]]) label("child")
        {
            // Update a[j][0] ... a[j][len[j]]
            // ...
        }
    }
}
```

Listing 4.5 Parent task with `auto` clause

Listing 4.6 is the first step in performance optimisation, where the programmer has declared that the subtasks of `parent` have unknown accesses, but they are all known to be in array `a[0;M]`. If a later task has strong access on some other region of memory, then knowing that there can be no ordering constraint that would require it to execute after a subtask of `parent` allows the tasks to be executed concurrently.

```c
#pragma oss task depend(auto : a[0; M]) label("parent")
{
    // ...
    for (int j = 0; j < N; j++)
    {
        #pragma oss task depend(inout : a[j][len[j]]) label("child")
        {
            // Update a[j][0] ... a[j][len[j]]
            // ...
        }
    }
}
```

Listing 4.6 Parent task with `auto` clause specifying a range

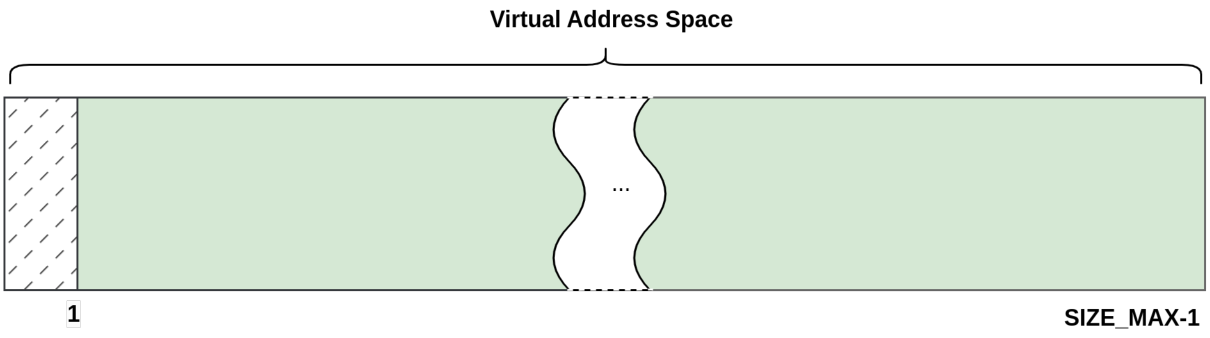
4.3 Programmer's Model

Our main extension to the programmer's model is the `auto` and `none` clauses. The `auto` clause, in fact, is an access type, similar to `in`, `out`, `inout` and so on.

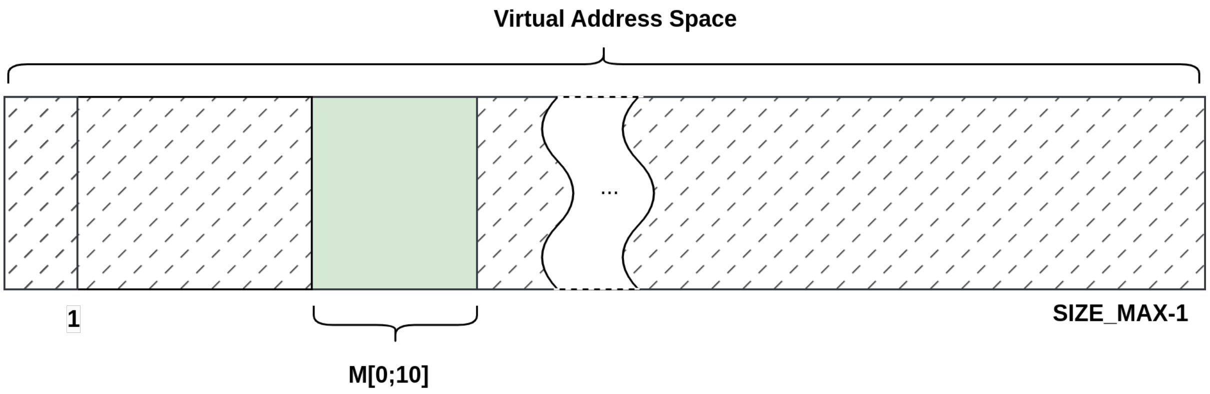
4.3.1 Auto Access Type

The semantics of the `auto` access type indicate the possible range of the virtual memory that the `auto` uses to infer the subtask accesses. Figure 4.2a illustrates this, where the highlighted green area represents the range from which `auto` will infer the subtask accesses. By default, it covers the whole virtual address space from 1 to `SIZE_MAX` − 1 inclusive.[1]

In certain cases, automatic aggregation may only be necessary for specific regions. These regions could include unknown subtask accesses within a known array, memory allocations within a known memory pool, or memory obtained from specially mapped memory regions. In such cases, specific regions can be defined for access using the syntax `depend(auto: addr[offset;size])` that narrows down the range of `auto` to that specific memory region. An illustration is shown in Figure 4.2b, which limits the `auto` range to the memory region `M[0;10]`. The syntax to achieve this is `depend(auto: M[0;10])`.



(a) Default `auto` range



(b) Using specific region for `auto` inference range.

Figure 4.2 Semantics of `auto` access type

[1]The access starts at 1 because accesses starting at `NULL` are ignored.

4.3.2 None Access Type

Additionally, it may be known that none of the descendent tasks of a task with the `auto` clause will access a specific region of memory, and some analysis show that excluding this region from the `auto` range is beneficial for the performance. We introduce `none` access type to indicate a region will not accessed by the current task (and/or its subtasks) and can safely assume that no accesses need to be inferred in that region beyond those explicitly expressed by the other task accesses. An example scenario is when a performance analysis shows that a later task that should execute concurrently is serialised after the current task. It happens when the later task has a strong access on a memory region and the runtime cannot know if the current task will create a subtask that accesses the same memory. This subtask would be ordered before the later task according to the sequential task ordering. Such serialisation can sometimes be solved by narrowing the scope of the `auto` access by specifying an access region as a `none` access region as indicated by region `M[0;10]` in Figure 4.3, and the code to achieve this is `depend(none: M[0;10])`.

This example was illustrated by the `row` task in Listing 4.3 in Section 4.2.1. Without the none clause, the runtime would have to assume that a row task might create subtasks that access elements of `C` other than `C[i][j]`. This assumption would force the runtime to serialize row tasks to avoid potential conflicts. By specifying none, we inform the runtime that such conflicts will not occur, allowing row tasks to execute concurrently.

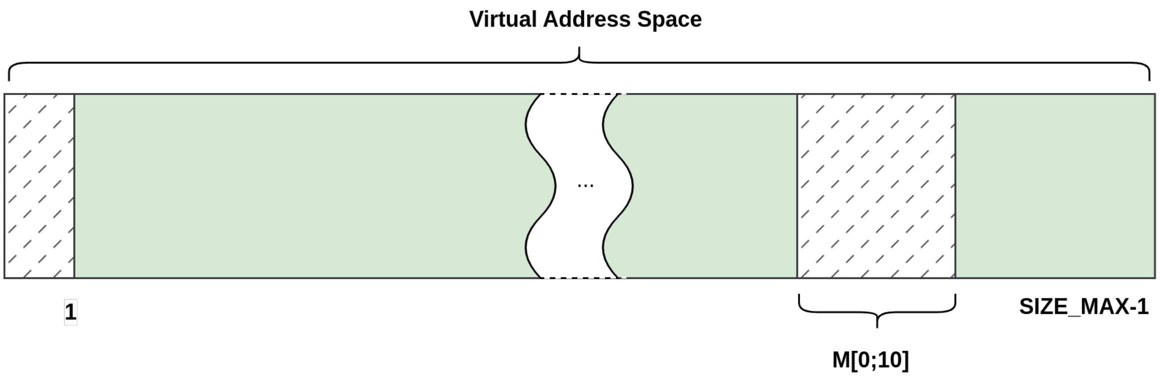


Figure 4.3 Semantics of `none` access type

Both `auto` and `none` syntaxes is similar to the syntax of usual access types such as `in`, `out`, and `inout`. The proposed syntax is given in Figure 4.4, which shows OpenMP-style and OmpSs-2-style data access specifications. A task with an `auto` clause still requires strong data accesses for the data that is accessed directly by the task. But it does not, in principle, require any explicit (weak or strong) accesses on behalf of its subtasks.

```
#pragma omp task depend(auto: <list>)
```

(a) Proposed OpenMP-style syntax

```
#pragma oss task auto
#pragma oss task auto(<region>)
```

(b) OmpSs-2-style syntax

Figure 4.4 Proposed OpenMP and OmpSs-2 syntax for `auto`

4.3.3 Upgrade Rules

As described in Section 2.1.2, specific access rules govern multiple accesses covering the same memory region. Table 2.1 presents the basic upgrade rules for ordinary types. Table 4.1 specifies how to resolve the situation for `auto` and `none` clauses. For example, a task that has both `depend(in: a)` (first row) and `depend(out: a)` (second column) is equivalent to one with the single access `depend(inout: a)`, since `inout` implies all necessary ordering constraints and data transfer requirements and there is no less restrictive data access type available. The combined access is strong if either access is strong. We extend to `auto` accesses by defining that `auto` is overridden by all other access types. So, for example, the combined effect of overlapping `auto` and `weakin` accesses is `weakin`. A `none` access that overlaps any access type other than `auto` adds no additional ordering or data transfer requirements, so it has no effect. But `none` is a specific access type that overrides `auto` type. The upgrade rules are commutative, so the combined effect of two accesses does not depend on the order in which the programmer writes them. The table is, therefore, symmetric, and only the upper triangle is shown. The upgrade rules are also associative, so the order in which the upgrade rules are applied is insignificant.

4.3.4 Fragmentation

It is valid in OmpSs-2, and therefore OmpSs-2@Cluster, for data accesses of the same or different tasks to partially overlap. The fragmented linear region dependencies, which are mandatory in the OmpSs-2 specification and the only supported dependency system for OmpSs-2@Cluster, will fragment (and unfragment) data accesses accordingly [27].

Table 4.1 Extended access upgrade rules of with the new `auto` and `none` clauses.

	in	**out**	**inout**	**concurrent**	**commutative**	**reduction**	**none**	**auto**
in	in	inout	inout	inout	inout	invalid	in	in
out	-	out	inout	inout	inout	invalid	out	out
inout	-	-	inout	inout	inout	invalid	inout	inout
concurrent	-	-	-	conc.	comm.	invalid	conc.	conc.
commutative	-	-	-	-	comm.	invalid	comm.	comm.
reduction	-	-	-	-	-	reduction*	red.	red.
none	-	-	-	-	-	-	none	none
auto	-	-	-	-	-	-	-	auto

* Non-identical overlapping reductions on the same task are undefined.

An `auto` access may initially cover a large part of the virtual address space, but it may be decomposed into subregions.

4.3.5 Inheritance of auto and none regions

If a task with the `auto` clause creates a subtask that, in turn, has one or more `auto` clauses, the subtask's clause will be restricted to cover most of the regions covered by parent accesses after the upgrade rules (other than `none`). This means that restrictions on the scope of analysis of a task and, by implication, its descendants, can be provided once at the the topmost level of the program, without duplicating this information throughout the codebase.

4.4 Implementation

4.4.1 Compiler

The only necessary change in the compiler is to add the new `auto` and `none` access types for task accesses. The compiler transformations for these access types are analogous to those for the existing `in`, `inout`, `out`, `concurrent` and `commutative` access types, the only difference being that an `auto` dependency is allowed to omit the access region. These two data access types have also been added to the Nanos6 API. While sophisticated compiler analysis could be used to narrow the scope of the `auto` access to reduce overhead and avoid serialisation, we have yet to find it necessary in our first implementation.

4.4.2 Runtime

4.4.2.1 Baseline Implementation

A strong advantage of this proposal is that there is a simple baseline implementation. Three things are required. First, the runtime must respect the extended access upgrade rules in Table 4.1. Secondly, it must remove all regions with `auto` access-type that are not part of the parent accesses. It must also downgrade `auto` accesses inside a parent's `in` access to `weakin`. It must do this to conform to the programmer's model and to ensure proper nesting of data accesses. Thirdly, it must treat any remaining `auto` accesses like `weakinout`. This is a valid implementation of the programming model. There is also no overhead for programs that do not have `auto` accesses. The overhead is already low for SMP (Section 4.2.2), but two optimisations are necessary for OmpSs-2@Cluster.

4.4.2.2 Optimising non-accessed regions when offloading tasks

The job of a parent task with an `auto` access is typically to create the subtasks that will do the computations. Most of the `auto` access(es) will likely not be needed for subtask accesses. Such regions can be identified when the parent task is completed, which is generally a long time before the data values are ready and off of the critical path. These accesses can be recognised as those for which the runtime system has not registered any subtask in the *bottom map*, the map from addresses to the currently-last subtask (if one exists) that is used to build the dependency graph. The OmpSs-2@Cluster runtime identifies such accesses and sends a message to the offloading node to prevent an unnecessary eager data send. When the region later becomes ready, there is usually a message to the remote node to indicate that it is ready and another message back, passing this information to the next task. As an optimisation, the offloading node skips this ping-pong, reducing the latency of the critical path.

4.4.2.3 Optimising read-only regions when offloading tasks

In OmpSs-2@Cluster, if a task has an `in` access, then as soon as it can read the data, this permission (read satisfiability) is immediately passed to the successor task, even if the task is offloaded without going via the remote node. It is only necessary to inform back to the offloader when the access has been completed. For an `auto` read-only access, this ability is only passed back to a successor on a different node when the access is complete. The solution is to send a notification similar to the non-accessed notification of Section 4.4.2.1. On receipt of this notification, the offloader sets a flag to indicate that read satisfiability can be immediately propagated to the next task.

4.5 Methodology and Benchmarks

4.5.1 Hardware and software platform

1. **Programming model**:
   We draw our study based on OmpSs-2@Cluster task-based parallel programming model and specifications (Section 2.2). We use Nanos6@Cluster runtime system as the reference implementation of the OmpSs-2@Cluster programming model (Section 2.3.2). In addition, the Mercurium source-to-source compiler is used to translate source code written with OmpSs-2 directives into a parallel taskfied version of the source code. Both Nanos6 and Mercurium are developed by the programming models group at BSC.

2. **Software tools**:
   We use Extrae [33] performance instrumentation tool for generating execution traces for detailed insights into the runtime implementation behaviour and the benchmark performance. Paraver [85] is the tool we use to analyse and visualise execution traces by Extrae. The Performance Tools Group developed the two tools at BSC. GNU Compiler Collection (GCC) 7.2.0 was used to compile all benchmarks and the modified runtime. The runtime uses Intel MPI 2018.4, which is the default and supported the implementation of MPI on MareNostrum, which fully exploits the 100 Gb/s Intel OmniPath network and Host Fabric Adapter (HFI) Silicon 100 series PCIe adaptor.
   The benchmarks use the Basic Linear Algebra Subprograms (BLAS) functions provided by Math Kernel Library (MKL) 2018.4. In addition, we used a regression testing framework developed by us, which automated and monitored the performance of the benchmarks between each stage of the modified Nanos6@Cluster runtime development. The Simple Linux Utility for Resource Management (SLURM) [86] was used as the cluster management and job scheduling system for handling nodes's resource allocation on MN4.

3. **Hardware equipment**:
   We evaluated our modified implementation of the Nanos6@Cluster and performed our experiments on up to 32 nodes of the general-purpose partition of the MareNostrum 4 supercomputer (MN4) [87]. MareNostrum 4 comprises 3456 compute nodes, each with two 24-core Intel Xeon Platinum 8160 sockets at 2.10 GHz, for a total of 48 cores per node. Each socket has a shared 32 MB L3 cache.

The memory capacity of nodes used have 96 GB physical memory (2 GB per core). The interconnect is 100 Gb/s Intel Omni-Path with a fat tree.

4.5.2 Benchmarks

Table 4.2 list the benchmarks that were used for evaluation of the `auto` type. The benchmarks represent common HPC applications used in the literature and depict computation patterns known to expose well-known and previously studied OmpSs-2@Cluster's performance issues [7, 19, 18]. All experiments reflect the best empirical blocksize or grainsize collected as a pre-experimenting step. In addition, all benchmarks are executed in configurations of 2 processes per node (one per NUMA node), following previous work [7], which found that using one process per socket led to better and less variable results due to the more effective use of NUMA locality.

All benchmark kernels were in separate source files, identical for all programming models and compiled with the same compiler flags. Each data point shows the average and standard deviation across ten runs for `auto` and five runs in `taskitr` case. Each run executes runs in different batch jobs and is confirmed to have different node allocations, which is essential to capture the randomness and variability in the cluster. In addition, the same batch job was used to test all programming models using the same node allocation to ensure fairness, meaning that for a single run, we test all variants of the same benchmark on the same allocation.

We evaluate the `auto` clause on SMP for three benchmarks, matrix multiplication (matmul-smp) and n-body (n-body-smp) benchmarks are adopted from the examples in [88] and the hypermatrix adopted from Section 4.2.1.

Since the overhead of our implementation on SMP is low, we also include more challenging benchmarks using OmpSs-2@Cluster evaluated for the best possible and similar configurations as Aguilar et al. [7]. multi-matvec is a sequence of identical dense double-precision matrix–vector multiplications, with the matrix distributed by rows and without dependencies between iterations. It has fine-grained tasks with complexity $O(n^2)$ and no inter-node data transfers. multi-matmul is a sequence of dense double-precision matrix–matrix multiplications, with larger $O(n^3)$ tasks and also no inter-node data transfers.

jacobi is an iterative double-precision Jacobi solver for dense, strictly diagonally dominant systems. It is equivalent to repeatedly pre-multiplying a vector by a dense square matrix. It has the same $O(n^2)$ complexity as multi-matvec, but an all-to-all communication pattern, making it a particularly good fit for fork–join parallelism.

cholesky is a Cholesky decomposition with a complex execution and dependencies pattern with complexity $O(n^3)/3$. This benchmark performs a higher number of smaller tasks, compared with matmul, and it introduces load imbalance and irregular patterns. The optimised version uses `task for` and memory reordering optimisations to reduce fragmentation and data transfers.

The n-body code [89] is an OmpSs-2@Cluster tasking implementation of Barnes–Hut [90]. The baseline implementation has a taskwait between constructing the tree and updating the particles. The optimised implementation uses an `auto` clause to replace the taskwait with a dependency.

Table 4.2 Evaluation benchmarks for the `auto` approach.

Benchmark	Parameters	Description
SMP:		
`hypermatrix`	$N = 16384$ to 55296	Hypermatrix matrix multiplication followed by Cholesky decomposition
`matmul-smp`	$N = 4096$	Matrix multiply without BLAS using nested weak and strong tasks [88]
n`-body-smp`	$N = 262144$	$O(n^2)$ n-body code with two nested loops to determine the forces on the particles [88]
Distributed memory:		
`multi-matvec`	$N = 65536$	Repeated dense matrix–vector multiplication using nested weak and strong tasks [7]
`multi-matmul`	$N = 32768$	Repeated dense matrix-matrix multiplication using nested weak and strong tasks [7]
`jacobi`	$N = 65536$	Jacobi iteration with nested weak and strong tasks [7]
`cholesky`	$N = 65536$	Cholesky decomposition with nested weak and strong tasks, task for, memory reordering and priority [7]
n`-body`	$N = 1000000$	Barnes–Hut using nested weak tasks and strong taskloops

4.6 Evaluation and Results

This section presents the results of the evaluated benchmarks, which are described in Section 4.5.2 and summarised in Table 4.2. We evaluate a subset of the benchmarks on SMP and on clusters up to 32 nodes, comparing the performance using `auto` versus using `taskwait`. In addition, we extend the evaluation for three variants on cluster up to 32 nodes as well: *manual*, *auto (unoptimised)*, and *auto (optimised)* that employs optimisations described in Section 4.4.2. The `manual` variant is adapted from the original implementation of [7], which uses nested tasks with the parent using weak dependencies, and the subtasks use strong dependencies. The `auto` (unoptimised) version is simply substituting all weak dependencies in the `manual` with `auto` clause, keeping everything else the same. In some cases, careful performance analysis and the adjusting of some of the dependencies accordingly can give a better performance representing a way of using the `auto` clause for obtaining a *first step optimised* version in a short time as in Figure 4.14. Listing 4.7 and 4.8 show part of the `cholesky` benchmark demonstrating an example of substituting weak dependencies with `auto` accesses, which is emphasised by the highlighted lines in the listing. We remark on the results by evaluating the productivity of the *manual* and *auto* variants as a function of the number of weak access substituted with auto access in the whole benchmark.

```
#pragma oss task node(node) label("weak_syrk")                    \
weakin( {A[idxk0 + i*blocks_per_node; npcols][0;ts][0;ts], i=0;pcols} ) \
weakinout(A[first; count][0;ts][0;ts])                            \
{
    for (size_t i = k + 1; i < nt; ++i) {
        int nodeii = get_block_node(&info, i, i);
        if (nodeii == node) {
            // ...
            #pragma oss task                              \
            in(Aki[0; ts][0; ts])                         \
            inout(Aii[0; ts][0; ts])                      \
            node(nanos6_cluster_no_offload) label("syrk")
            oss_syrk(ts, Aki, Aii, k, i, i, 0);
        }

        for (size_t j = k + 1; j < i; ++j) {
            int nodeji = get_block_node(&info, j, i);
            if (nodeji == node) {
                // ...
                #pragma oss task                              \
                in(Aki[0; ts][0; ts])                         \
                in(Akj[0; ts][0; ts])                         \
                inout(Aji[0; ts][0; ts])                      \
                node(nanos6_cluster_no_offload) label("gemm")
                oss_gemm(ts, Aki, Akj, Aji, k, j, i, 0);
            }
        }
    }
}
```

Listing 4.7 Example showing part of `cholesky` benchmark `manual` implementation using nested tasks.

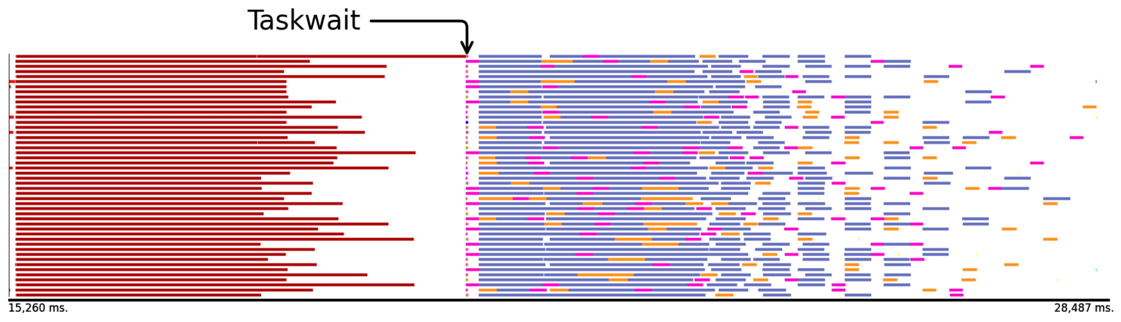
```
#pragma oss task auto nowait node(node) label("auto_syrk")
{
    for (size_t i = k + 1; i < nt; ++i)
    {
        int nodeii = get_block_node(&info, i, i);
        if (nodeii == node) {
            // ...
            #pragma oss task                              \
            in(Aki[0; ts][0; ts])                         \
            inout(Aii[0; ts][0; ts])                      \
            node(nanos6_cluster_no_offload) label("syrk")
            oss_syrk(ts, Aki, Aii, k, i, i, 0);
        }

        for (size_t j = k + 1; j < i; ++j) {
            int nodeji = get_block_node(&info, j, i);
            if (nodeji == node) {
                // ...
                #pragma oss task                              \
                in(Aki[0; ts][0; ts])                         \
                in(Akj[0; ts][0; ts])                         \
                inout(Aji[0; ts][0; ts])                      \
                node(nanos6_cluster_no_offload) label("gemm")
                oss_gemm(ts, Aki, Akj, Aji, k, j, i, 0);
            }
        }
    }
}
```

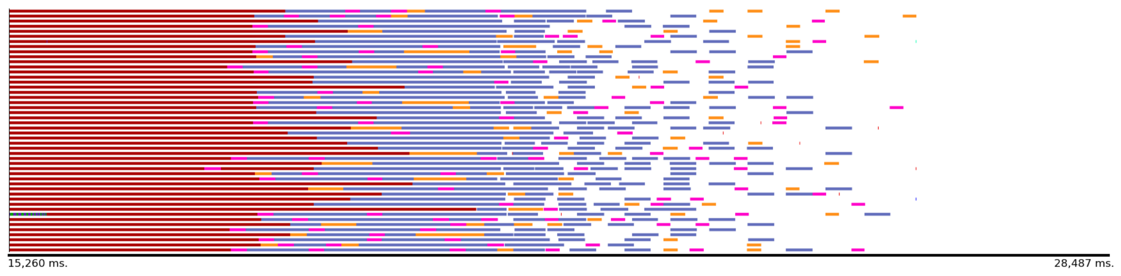
Listing 4.8 Example showing part of `cholesky` benchmark where the weak accesses in Listing 4.7 are substituted with `auto` access.

4.6.1 SMP Evaluation

We start by showing the performance when using `auto` compared to `taskwait` for the `hypermatrix` example program from Section 4.2.1. The performance is visualised as an execution trace collected with Extrae tracing tool [33] and visualised with Paraver performance visualizer tool [85]. Figure 4.5a shows the `taskwait`, which is similar to the trace already shown in 4.2.1, and Figure 4.5b shows the `auto` performance. The trace is for a problem size of $N = 30720$, which corresponds to an upper triangular matrix of 3.5 GB, on all 48 cores of a single node on the MN4. In Figure 4.5a, we see that all `matmul` tasks must be completed before the Cholesky decomposition can start. This synchronisation is due to the taskwait on line 21 of the program in Listing 4.4. Since the number of `matmul` tasks is different for different blocks of the matrix multiplication, the loads on the cores are not perfectly balanced. The Cholesky decomposition involves the `trsm`, `gemm`, `syrk` tasks, as well as `potrf`, which is too small to see, and several weak tasks, which are also too small to see. In Figure 4.5b, we see the effect of the `auto` clause in correctly specifying the precise task dependencies without needing a taskwait or sentinel. The traces match the motivational example in section 4.2.1. Both traces use the same x-axis scale and for the same time duration.



(a) Using taskwait



(b) Using `auto` clause

🟥 matmul 🟪 trsm 🟦 gemm 🟧 syrk

Figure 4.5 Extrae/Paraver trace of sparse hypermatrix benchmark. Both variants have precise data access on all tasks. Subfigure (a) has a taskwait and subfigure (b) uses the `auto` clause, which avoids the synchronization after the matrix multiplication. The execution time of `potrf` is very short, so these tasks are not visible in the traces.

Figure 4.6 shows the TFLOP/s obtained for `hypermatrix`, as the problem size is varied between $N = 16384$ (1 GB) and $N = 55296$ (11.4 GB). All data points use all 48 cores of a single MareNostrum 4 node. They are the average of five executions; the standard deviation is $< 1\%$ in all cases. For the larger problem sizes, there is a roughly 5% performance increase. While this improvement is not enormous, it does demonstrate the potential. It is limited by the critical path of the final part of the Cholesky decomposition and could likely be improved using a better task scheduling policy.

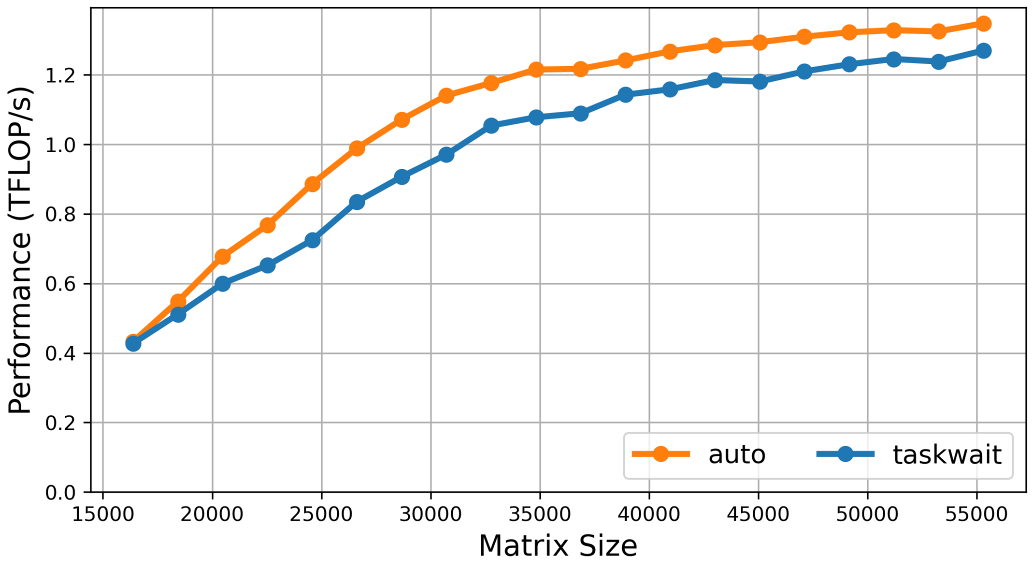


Figure 4.6 Performance of `auto` and taskwait versions of sparse hypermatrix on 1 node

Figure 4.7 shows the performance of the `matmul-smp`. We see that across the whole range of block sizes, the simpler implementation using `auto` is always within 19.6% of the original "manual" version.

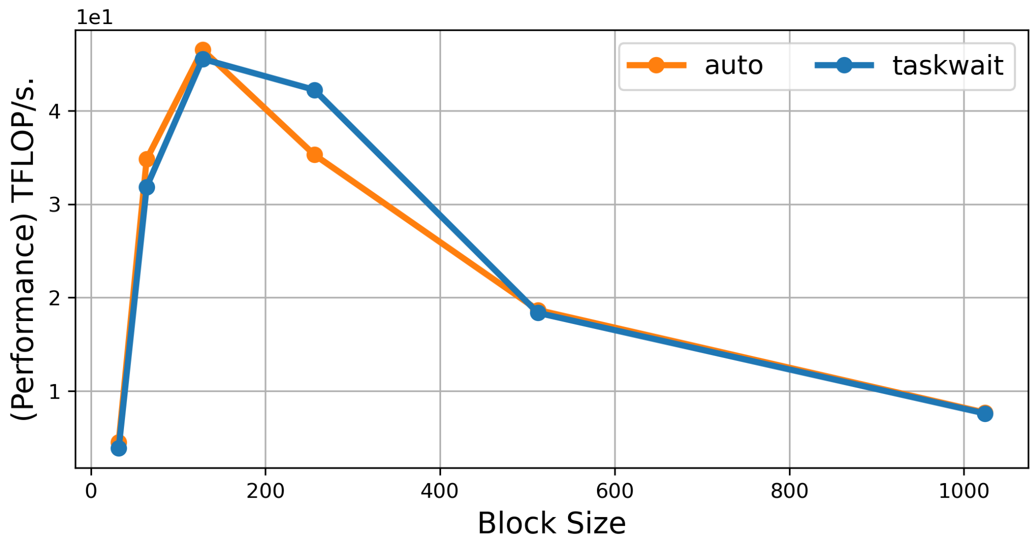


Figure 4.7 Throughput of the `matmul-smp` benchmark on 1 node for the same problem size with different block sizes. The version with `auto` to deduce all weak accesses is within 19.6% of the original `manual` version.

Similarly Figure 4.8 show the n-`body-smp` performance on a single node, reporting 10% of the of the original "manual" version as well

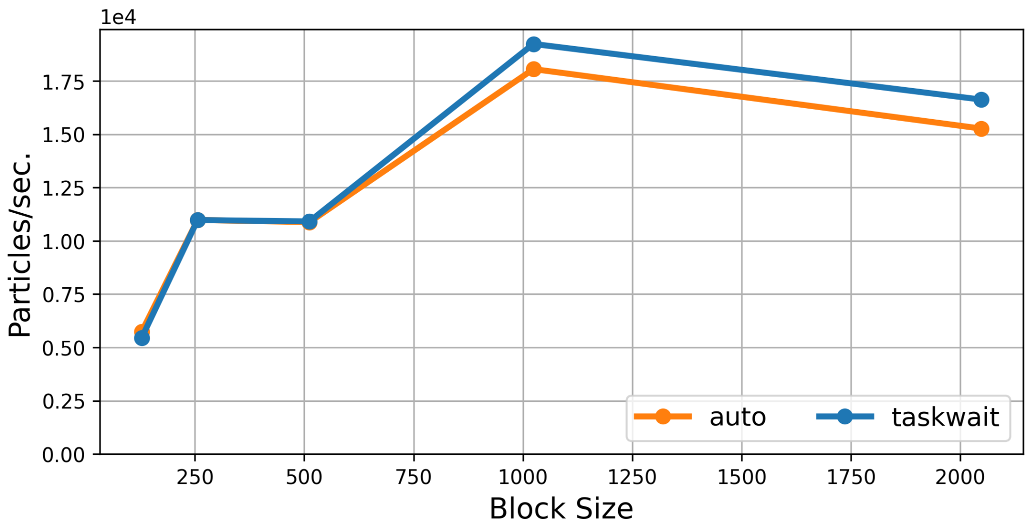


Figure 4.8 Throughput of the `n-body-smp` benchmark on 1 node for the same problem size with different block sizes. The version with `auto` to deduce all weak accesses is within 10% of the original `manual` version.

4.6.2 Cluster Evaluation (auto vs. taskwait)

Figure 4.9 shows the performance of the `auto` clause for n-body, with strong scaling on 1 to 32 nodes with 1 process per node. Both versions build the tree of unknown size, using a taskloop with a commutative dependency on the tree, and another task calculates the forces and updates the particles using a taskloop. The taskloop is automatically distributed among the nodes, so the tree size must be correct to avoid copying too much data. The taskwait version needs a taskwait to obtain the size of the tree, whereas the `auto` version encloses the taskloop in a parent task with an `auto` data access. We see in Figure 4.9 that the `auto` version has consistently higher throughput than the manual version with taskwaits, at 195,000 particles per second on 16 nodes, compared with 137,000 particles per second on 16 nodes for the manual version with taskwaits. This is a 1.4 times increase in throughput.

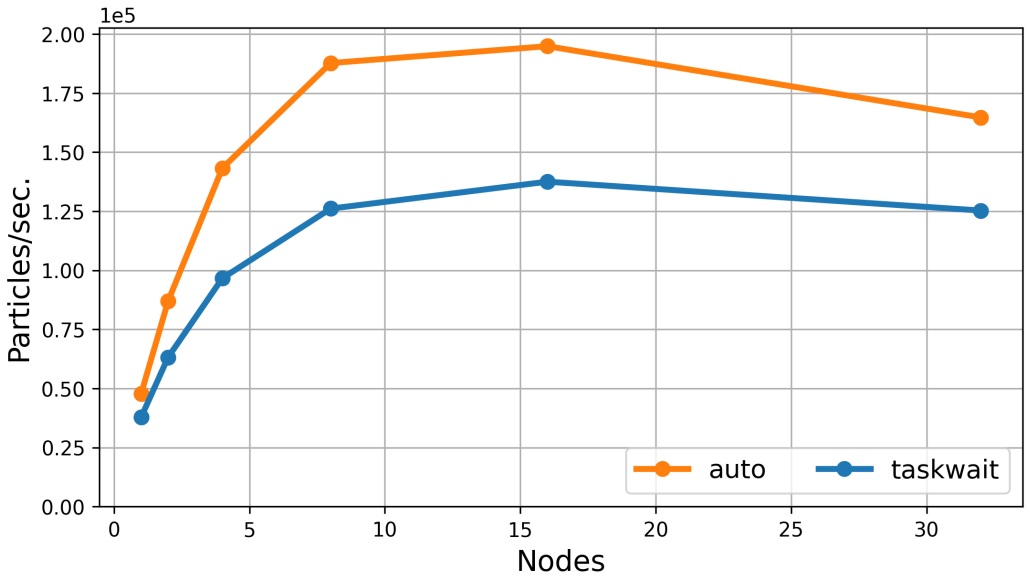
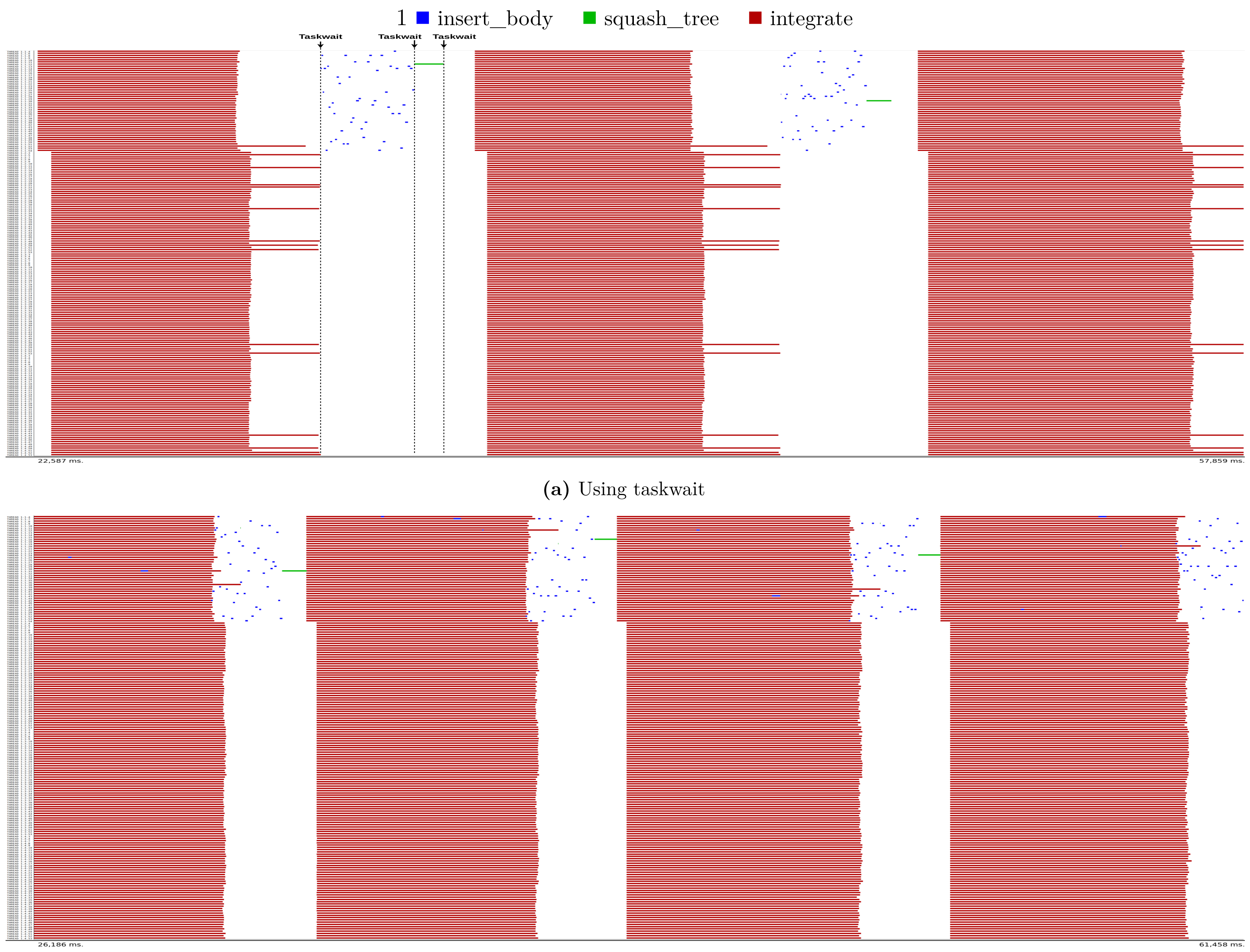


Figure 4.9 Strong scaling of `auto` and taskwait versions of n-body on 1 to 32

An execution trace of the n-body benchmark on 4 nodes is shown in Figure 4.10, which was collected and visualized similarly to the trace shown in Section 4.6.2. The trace is for 10 iterations of the same problem size (1M particles) as in the results shown in Figure 4.9. Both variants are shown starting from the beginning of the same iteration and for the same duration. Both implementations create the tree locally on `node 0`; this is denoted by the `insert_body` tasks. In addition, both implementations optimize the memory access by "squash" or flattening the tree structure into a contiguous array of cells, indicated by the `squash_tree` task. Finally, the force integration part (`integrate` tasks) of the bodies in the tress is done using `taskloop` iterative construct that automatically distributed among the node depending on grainsize [9] which controls the size or amount of work assigned to each task in terms of number of loop iterations.

The `taskwait` variant uses 3 `taskwait` indicated by the dashed vertical lines in the trace Figure 4.10a, after constructing the tree to obtain the size of the tree, and before integrating the forces for the new iteration to make sure the data is squashed, and between individual iterations. However, the `auto` variant does not need any `taskwait`, which uses two weak tasks with nested strong subtasks. One weak task allocates the tree, builds it, and squashes it. The second weak task performs the force integration also with taskloop similar to the `taskwait` variant and using the same grainsize. It is clearly seen that the `auto` variant can start creating the integration tasks immediately without waiting as in the `taskwait` case, allowing for more parallelism.



(a) Using taskwait

(b) Using `auto` clause

Figure 4.10 Extrae/Paraver trace of n-body benchmark on 4 nodes for the same time duration.

4.6.3 Cluster Evaluation (auto vs. manual)

This section shows the performance of the `auto` clause for the four benchmarks, `multi-matvec`, `multi-matmul`, `jacobi`, and `cholesky`, with strong scaling from 1 to 32 nodes. In all of the results, The x-axis is the number of nodes, each with two processes. The y-axis is the aggregate performance in GFLOP/s. As mentioned earlier, we evaluate the performance for three variants: *manual*, *auto (unoptimised)* and *auto (optimised)*. The *manual* are the benchmarks in Aguilar et al. [7], which have complete and precise weak accesses specified manually for each parent task. The *auto (unoptimised)* and *auto (optimised)* versions both have all of the weak accesses replaced by the `auto` clause, with the default region covering the full address space (see Section 4.3.1).

In the *manual* variants, we were able to fairly closely (within about 4%) reproduce the results in Aguilar et al. [7] that were initially evaluated for 1 to 16 nodes, and we extended the evaluation to 32 nodes. In all cases, the `auto` (unoptimised) results are within 3% of *manual* on a single node, but the performance drops significantly on more than one node. Despite the performance drop, the `auto` (optimised) results provide a reasonable scaling level for obtaining the first functional version of a program with nested tasks in the shortest possible time with minimum effort as a productivity tool.

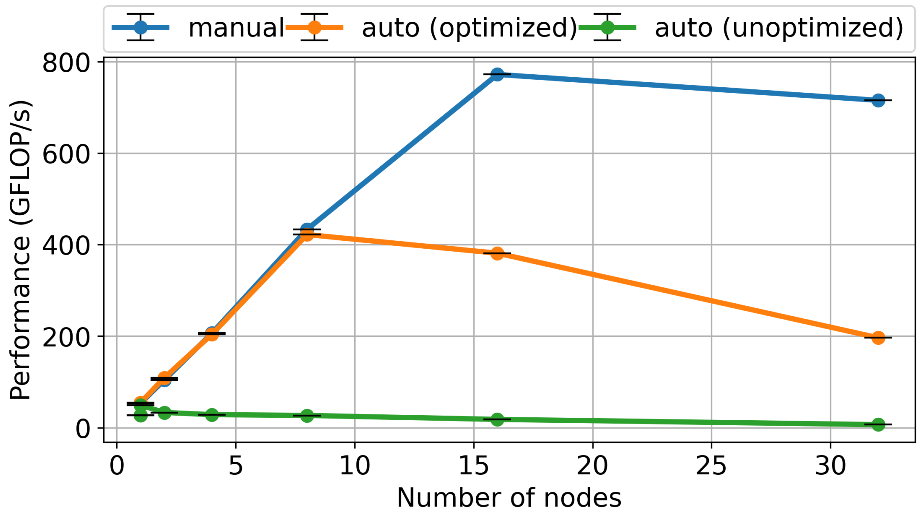


Figure 4.11 Strong scaling performance for `multi-matvec` benchmark using OmpSs-2@Cluster on 1 to 32 nodes.

Figure 4.11 shows 3% performance degradation for `multi-matvec` on up to 8 nodes, compared with the manual version. Similarly, Figure 4.12 shows 9% degradation on 8 nodes and 37% degradation on 16 nodes, again compared with the manual version.

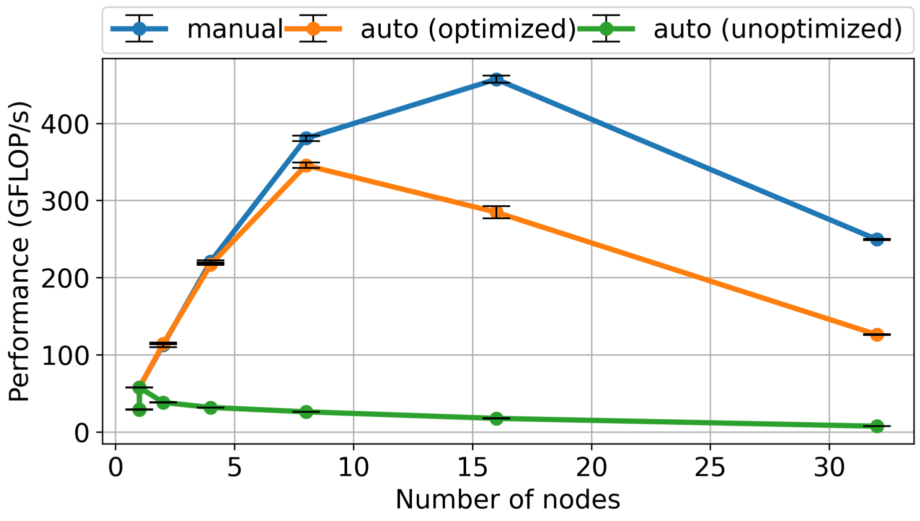


Figure 4.12 Strong scaling performance for `jacobi` benchmark using OmpSs-2@Cluster on 1 to 32 nodes.

Figure 4.13 show a matching performance of the `multi-matmul` benchmark for the `manual` and `auto` (optimised) with some variances due to noises indicated by the larger error-bar.

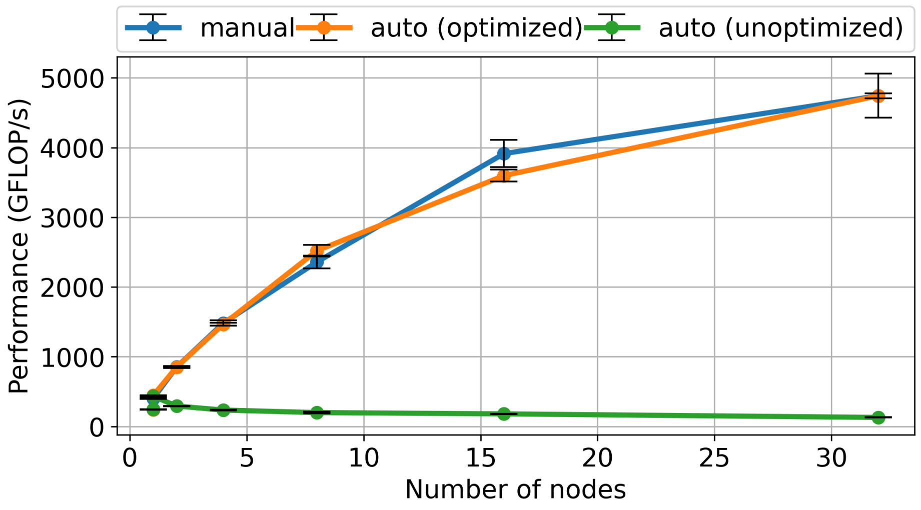


Figure 4.13 Strong scaling performance for `multi-matmul` benchmark using OmpSs-2@Cluster on 1 to 32 nodes.

Figure 4.14 shows the `cholesky` benchmark scaling only up to 4 nodes. Performance analysis using Extrae/Paraver showed excessive control message communication before one of the three offloaded tasks. Adding the true weak accesses to that task enables scaling on up to 8 nodes with less than 36% degradation from the original "manual" version. The red curve indicates this result "`first step opt`" in Figure 4.14 as a first step optimisation toward more efficient implementation in future.

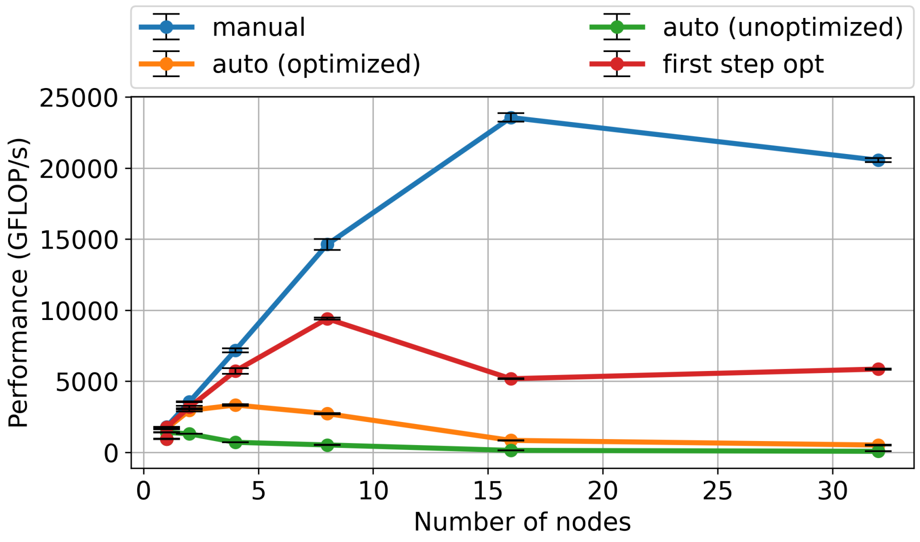


Figure 4.14 Strong scaling performance for `cholesky` benchmark using OmpSs-2@Cluster on 1 to 32 nodes.

4.6.4 Quantifying productivity

Quantifying `auto` productivity is challenging because the benefits of using `auto` are not directly captured by common metrics such as the number of lines of code, which is easily understandable and independent of the programming language [91]. As a more convenient metric to the nature of how `auto` is utilized, in this section, we quantify the productivity of `auto` based on the total number of `weak` accesses substituted by `auto` accesses in the entire benchmark.

This approach relies on the fact that `manual` and `auto` implementations are similar, differing only in the `weak/auto` accesses at the top-level tasks of any nested levels. Strong accesses are not considered since the canonical form of nested tasks in OmpSs-2@Cluster requires specifying memory regions accessed directly by top-level tasks as strong accesses. Hence, both `manual` and `auto` implementations will have the same strong accesses, if any. Table 4.3 summarizes the productivity of all evaluated benchmarks in this chapter. The table shows the total number of `weak` accesses present in all tasks of the manual implementation for each benchmark, as well as the number of `auto` accesses required to replace these `weak` accesses to obtain the `auto` implementation for the same benchmark.

Table 4.3 Productivity as a function of number of accesses

Benchmark	***weak*** **accesses**	***auto*** **accesses**
jacobi	9	4
multi-matvec/multi-matmul/matmul-smp	3	1
cholesky	7	4
n-body-smp	3	2

4.7 Conclusion

In this chapter, we propose the `auto` clause, which indicates that the task annotations may be incomplete due to unspecified subtask memory accesses or memory allocation. The `auto` clause allows a task to be created before the data accesses of the task and its descendants are known. Existing approaches need to either block using a taskwait or substitute "fake" accesses known as sentinels. As there is no need to block, our approach enables concurrent task creation and execution to continue without interruption, maintaining parallelism and affording maximum freedom to the scheduler to optimise load balance and data locality. Since task annotations can match the actual data accesses, a single mechanism controls task ordering, program data transfers on distributed memory, and optimises data locality. The `auto` clause also provides an incremental path to develop programs with nested tasks because an initial functional implementation can be created without the time-consuming and error-prone specification of weak accesses on all parent tasks. We present a straightforward runtime implementation with a few key optimisations. We evaluate our approach using a hypermatrix multiplication followed by Cholesky decomposition and a Barnes–Hut n-body application, which achieves a 1.4 times speedup on 32 nodes. We evaluate programmer productivity by replacing all weak accesses by `auto` on two SMP benchmarks, and four OmpSs-2@Cluster benchmarks show a $< 4\%$ slowdown for three of the benchmarks on 8 nodes.

CHAPTER 5

Distributed taskiter

5.1 Introduction

Iterative data-flow applications employ iterative methods or multi-step simulation techniques that usually involve dividing the iteration space into a series of tasks, each executing a subset of the iteration space in parallel. In distributing tasking with OmpSs-2@Cluster, iterative applications follow the same structure for distributing work across multiple nodes with the nested tasking approach described in Section 2.3.4. Listing 5.1 shows an iterative version of the example explained in Section 2.3.5.

```
for (int it = 0; it < NUM_ITERATIONS; ++it) {
    // Task A
    #pragma oss task node(1) depend(weakout:x)
    { ... }

    // Task B
    #pragma oss task node(2) depend(weakin:x)
    { ... }
}
```

Listing 5.1 OmpSs-2@Cluster iterative version of example in Listing 2.1.

Figure 5.1 illustrates the MPI control messages between nodes in this case, resembling those of a typical offloaded OmpSs-2@Cluster application described in Section 2.3.5. However, these messages are repeated for each executed iteration, resulting in a large number of control messages between the nodes as the number of iterations increases.

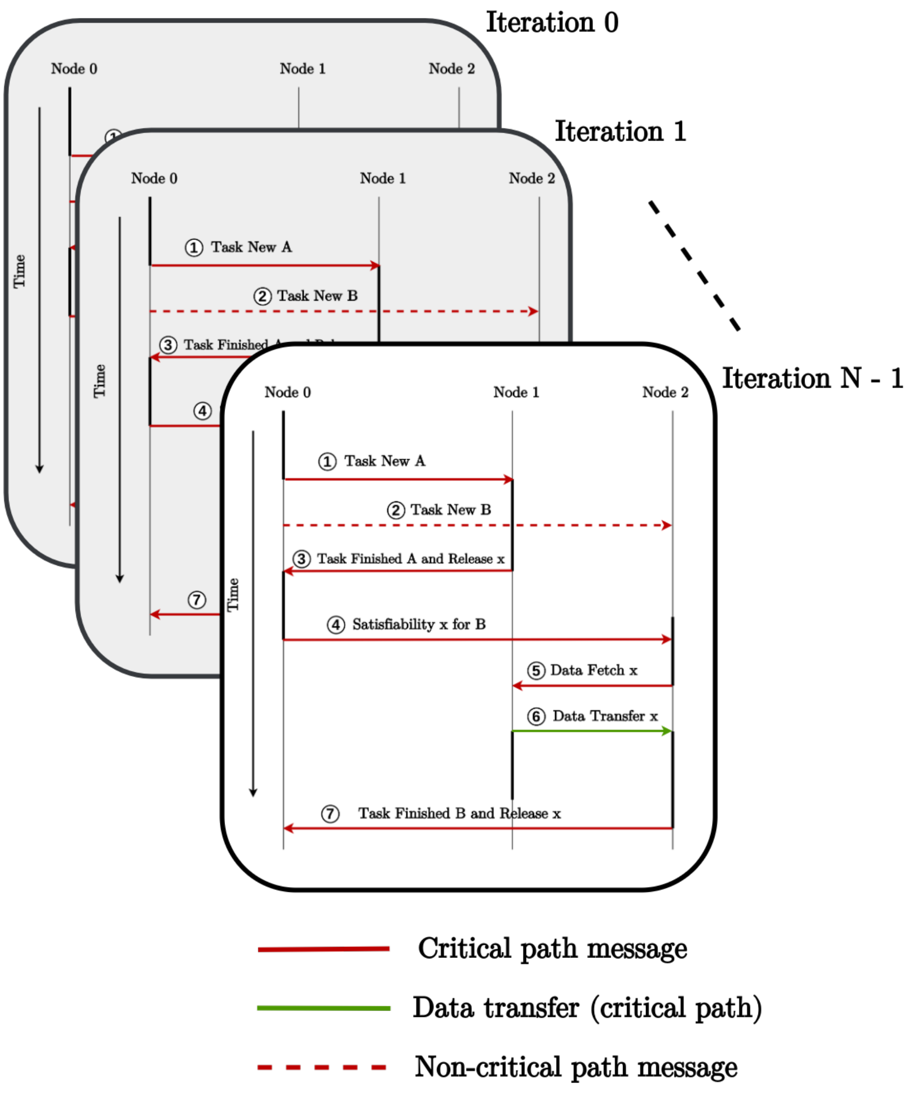


Figure 5.1 Control messages for OmpSs-2@Cluster iterative application.

This chapter utilizes the `taskiter` iterative construct originally proposed in [1], and defined in Section 2.4 to eliminate the excessive control messages shown in Figure 5.1 by leveraging the repetitive nature of HPC iterative applications that build the same DAG at each iteration or timestep in multi-step simulation applications.

The control messages between nodes, in this case, are as follows:

- "`Task New`" messages for offloading copies of the original `taskiter` to all nodes in the cluster.

- "`Data Transfer`" messages for transferring the required data at each iteration.
- "`Task Finished`" messages sent when an offloaded task completes all iterations, which are sent back to Node 0 to release the accesses.

We refer to our implementation as " distributed `taskiter` " to distinguish it from the originally proposed `taskiter` in [1]. Figure 5.2 illustrates the distributed `taskiter` control messages, emphasizing the significant difference between the control messages in the original OmpSs-2@Cluster iterative application (shown in Figure 5.1) and those in the `taskiter` approach. The only message overhead occurs at the beginning of offloading the `taskiter` task to all nodes via the "`Task New`" messages and at the end of execution via the "`Task Finished`" messages.

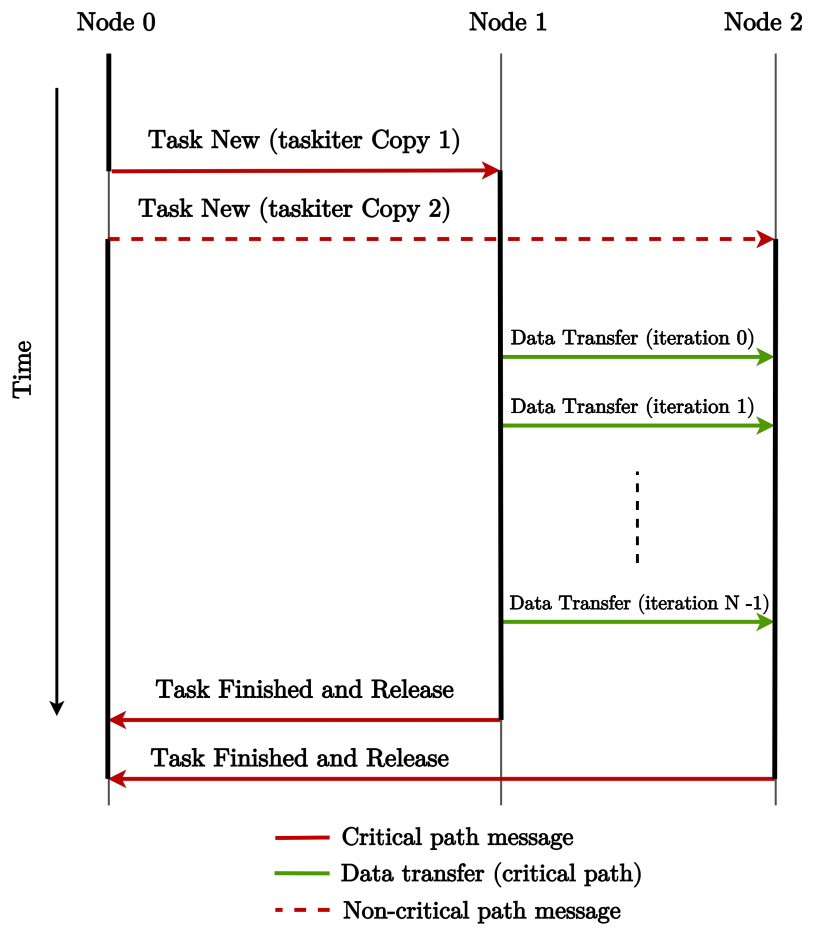

Node 0
Node 1
Node 2
Time
Task New (taskiter Copy 1)
Task New (taskiter Copy 2)
Data Transfer (iteration 0)
Data Transfer (iteration 1)
Data Transfer (iteration N -1)
Task Finished and Release
Task Finished and Release
Critical path message
Data transfer (critical path)
Non-critical path message

Figure 5.2 Distributed taskiter control messages.

5.2 Motivation

This section motivates this part of our work through three variants of a Gauss–Seidel 2D heat equation for three different programming models:fork–join MPI + OpenMP tasks, asynchronous TAMPI + OpenMP/OmpSs-2 tasks and OmpSs-2@Cluster with distributed taskiter, shown in Listing 5.2, 5.3, and 5.4 respectively. The benchmark is an in-place 2D stencil calculation where each element is updated based on the values above and to the left from the current timestep and the values to the right and below from the previous timestep. The matrix is a grid of NBY×NBX blocks, each of size BSY×BSX elements. It is distributed among the processes cyclically by rows, which is hard-coded in the case of the MPI+OpenMP and TAMPI+OpenMP/OmpSs-2 variants.

Listing 5.2 shows an implementation using fork–join parallelism with MPI and OpenMP. The local part of the stencil calculation has NBY_LOCAL rows, including the single-row halos at the top and bottom. The computation is done using tasks with dependencies (lines 35 to 49 in Listing 5.2) to update all elements using 2D wavefront parallelism in a timestep. The parallelism rises from zero at the beginning of the timestep up to a maximum of the number of blocks along the shortest dimension. It then drops back down to zero at the end of the timestep due to the taskwait on line 50.

```c
double matrix[NBY_LOCAL][NBX][BSY][BSX];
int main(int argc, char **argv)
{
    int provided;
    MPI_Init_thread(argc, argv, MPI_THREAD_MULTIPLE, &provided);
    assert(provided == MPI_THREAD_MULTIPLE);
    ...
    for (int it = 0; it < NUM_ITERATIONS; it++)
    {
        MPI_Request request[NBX * 3];
        int count = 0;
        if (rank != 0)
        {
            // Send first compute row
            for (x = 1; x < NBX - 1; x++)
            {
                MPI_Isend(&matrix[1][x][0][BSY - 1], BSX, MPI_DOUBLE, rank - 1, bx + it * NBX,
                    MPI_COMM_WORLD, &request[count++]);
            }
            // Receive upper border
            for (x = 1; x < NBX - 1; x++)
            {
                MPI_Irecv(&matrix[0][x][0][BSY - 1], BSX, MPI_DOUBLE, rank - 1, bx + it * NBX,
                    MPI_COMM_WORLD, &request[count++]);
            }
        }
        if (rank != rank_size - 1)
        {
            // Receive lower border
            for (x = 1; x < NBX - 1; x++)
            {
                MPI_Irecv(&matrix[NBY_LOCAL - 1][x][0], BSX, MPI_DOUBLE, rank + 1, bx + it *
                    NBX, MPI_COMM_WORLD, &request[count++]);
            }
        }
        MPI_Waitall(count, request, MPI_STATUSES_IGNORE);

        for (int y = 1; y < NBY_LOCAL - 1; y++)
        {
            for (int x = 1; x < NBX - 1; x++)
            {
                #pragma omp task                \
                depend(in : matrix[y - 1][x])   \
                depend(in : matrix[y][x - 1])   \
                depend(in : matrix[y][x + 1])   \
                depend(in : matrix[y + 1][x])   \
                depend(inout : matrix[y][x])
                {
                    GaussSeidelBlock(matrix, x, y);
                }
            }
        }
        #pragma omp taskwait
        if (rank != rank_size - 1)
        {
            // Send last compute row
            count = 0;
            for (x = 1; x < NBX - 1; x++)
            {
                MPI_Isend(&matrix[NBY_LOCAL - 2][x][0], BSX, MPI_DOUBLE, rank - 1, bx + it *
                    NBX, MPI_COMM_WORLD, &request[count++]);
            }
            MPI_Waitall(count, request, MPI_STATUSES_IGNORE);
        }
    }
}
```

Listing 5.2 Fork–join MPI + OpenMP tasks of Gauss–Seidel 2D heat equation.

Listing 5.3 shows an asynchronous TAMPI + OpenMP implementation. This version eliminates the taskwait between timesteps, and it has higher performance due to 3D wavefront parallelism, which allows concurrent execution of tasks from different timesteps. The parallelism rises from zero at the beginning of the first timestep, and it stays at the maximum value until close to the end of the last timestep. The cost is extra complexity in encapsulating the sends and receives into tasks and using TAMPI's non-blocking API.

Listing 5.4 shows the OmpSs-2@Cluster implementation using taskiter. Whereas most of the MPI + OpenMP and TAMPI + OpenMP versions' code concerns orchestration and micromanagement of data distribution and communication, only two declarative pieces of information have been introduced to the OmpSs-2@Cluster version. Firstly, the call to `nanos6_set_affinity` on line 5 describes the data affinity. This call does not move data, but it hints to the runtime that the non-readonly rows of the matrix ought to be distributed cyclically across the ranks. Secondly, the access to the blocks above and below the current block has been made more precise since the task only reads a single row of these blocks rather than the whole block.[1] The MPI + OpenMP and TAMPI + OpenMP/OmpSs-2 versions correctly perform data transfers of a single row of each block, i.e, each of size `BSX` elements, since that is the only data accessed by the neighboring rank. Without this change to the distributed taskiter version, the oversized accesses on the tasks would mislead the runtime, forcing it to send whole blocks, each of size `BSY`×`BSX` elements. We employ the fragmented regions dependency system [36], enabled by default in OmpSs-2@Cluster, allowing correct enforcement of dependencies between tasks having accesses to full and partial blocks.

Overall, the number of lines of code in the OmpSs-2@Cluster version with distributed taskiter is about one-third that of the TAMPI + OpenMP version and almost all of the code relates to the actual Gauss–Seidel computation.

[1]It is unnecessary to increase the precision of the blocks to the left and to the right due to the data distribution by rows among the nodes. But doing so is straightforward using multidependencies [92] and would not introduce any overheads during execution.

```c
double matrix[NBY_LOCAL][NBX][BSY][BSX];
int main(int argc, char **argv)
{
    int provided;
    MPI_Init_thread(argc, argv, MPI_TASK_MULTIPLE, &provided);
    assert(provided == MPI_TASK_MULTIPLE);
    ...
    for (int it = 0; it < NUM_ITERATIONS; it++)
    {
        if (rank != 0)
        {
            // Send first compute row
            for (x = 1; x < NBX - 1; x++)
            {
                #pragma omp task depend(in : matrix[1][x])
                {
                    MPI_Request request;
                    MPI_Isend(&matrix[1][x][0][BSY - 1], BSX, MPI_DOUBLE, rank - 1, bx + it *
                        NBX, MPI_COMM_WORLD, &request);
                    TAMPI_Iwait(&request, MPI_STATUS_IGNORE);
                }
            }
            // Receive upper border
            for (x = 1; x < NBX - 1; x++)
            {
                #pragma omp task depend(out : matrix[0][x])
                {
                    MPI_Request request;
                    MPI_Irecv(&matrix[0][x][0][BSY - 1], BSX, MPI_DOUBLE, rank - 1, bx + it *
                        NBX, MPI_COMM_WORLD, &request);
                    TAMPI_Iwait(&request, MPI_STATUS_IGNORE);
                }
            }
        }
        if (rank != rank_size - 1)
        {
            // Receive lower border
            for (x = 1; x < NBX - 1; x++)
            {
                #pragma omp task depend(out : matrix[0][x])
                {
                    MPI_Request request;
                    MPI_Irecv(&matrix[NBY_LOCAL - 1][x][0], BSX, MPI_DOUBLE, rank + 1, bx + it
                        * NBX, MPI_COMM_WORLD, &request);
                    TAMPI_Iwait(&request, MPI_STATUS_IGNORE);
                }
            }
        }
        for (int y = 1; y < NBY_LOCAL - 1; y++)
        {
            for (int x = 1; x < NBX - 1; x++)
            {
                #pragma omp task              \
                depend(in : matrix[y - 1][x]) \
                depend(in : matrix[y][x - 1]) \
                depend(in : matrix[y][x + 1]) \
                depend(in : matrix[y + 1][x]) \
                depend(inout : matrix[y][x])
                {
                    GaussSeidelBlock(matrix, x, y);
                }
            }
        }
        if (rank != rank_size - 1)
        {
            // Send last compute row
            for (x = 1; x < NBX - 1; x++)
            {
                #pragma omp task depend(in : matrix[0][x])
                {
                    MPI_Request request;
                    MPI_Isend(&matrix[NBY_LOCAL - 2][x][0], BSX, MPI_DOUBLE, rank - 1, bx + it
                        * NBX, MPI_COMM_WORLD, &request);
                    TAMPI_Iwait(&request, MPI_STATUS_IGNORE);
                }
            }
        }
    }
}
```

Listing 5.3 Asynchronous TAMPI + OpenMP/OmpSs-2 of Gauss–Seidel 2D heat equation.

```
double matrix[NBY][NBX][BSY][BSX];
int main(int argc, char **argv)
{
    ...
    nanos6_set_affinity(&matrix[1], (NBY - 2) * NBX * BSY, BSX, nanos6_equpart_distribution, 0,
         NULL);
    #pragma oss taskiter depend(weakinout : matrix)
    for (int it = 0; it < NUM_ITERATIONS; it++)
    {
        for (int y = 1; y < NBY - 1; y++)
        {
            for (int x = 1; x < NBX - 1; x++)
            {
                #pragma oss task                          \
                depend(in : matrix[y - 1][x][BSY - 1])   \
                depend(in : matrix[y][x - 1])            \
                depend(in : matrix[y][x + 1])            \
                depend(in : matrix[y + 1][x][0])         \
                depend(inout : matrix[y][x])
                {
                    GaussSeidelBlock(matrix, x, y);
                }
            }
        }
    }
}
```

Listing 5.4 OmpSs-2@Cluster taskiter of Gauss–Seidel 2D heat equation.

5.3 Programmer's Model

The programmer's model for distributed taskiter is the same as taskiter on SMP [1], except for the rules related to the definition of accesses:

1. **Full definition of accesses:**

   OmpSs-2@Cluster requires all tasks to have a full specification of their accesses [7], so the runtime can program any necessary data transfers. This requirement is inherited for taskiters, and it differs from the situation on SMP, where the taskiter only needs to be given accesses when necessary to enforce ordering with its sibling tasks. Any data only required by subtasks should be specified as a weak access (defined in Section 2.1.2). Any data required by the loop body or loop condition needs to be specified as a strong (i.e., non-weak) access.

2. **Precise definition of accesses:**

   The task accesses give a unified specification of the data accessed by the task, both for task ordering and to program data transfers. These accesses should precisely define the data that is needed by the task in order to avoid unnecessary data transfers (an example was given in Section 5.2).

5.4 Implementation

The execution of a distributed taskiter is illustrated in Figure 5.3, which shows a timeline, from left to right, of the steps, ⓪, ①, ②, $\cdots$, ⑥, executed by each rank. This figure is intended to give an overview of the process rather than quantifying the relative durations of these steps, which are not to scale.

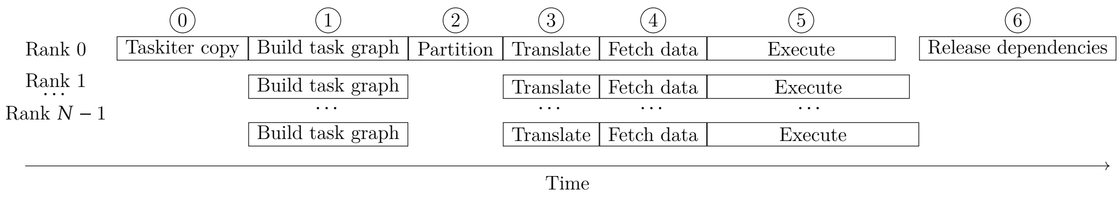


Figure 5.3 Illustration of the execution process for a distributed taskiter. This figure gives an overview of the sequence of steps and where they are executed, but it does not quantify the relative durations of the steps, which are not to scale.

5.4.1 Compilation

Compilation is done in the same way as taskiter on SMP, as the programmer's model in Section 5.3 does not require any changes to the compiler. The compiler encapsulates the loop body as a task, similar to a taskloop or taskfor. The taskiter task is passed to the runtime alongside the loop bounds and a flag to identify it as a taskiter.

5.4.2 Building The Taskiter Graph

The taskiter becomes ready following the same condition as any other task, i.e. as soon as all of its strong accesses, if any, are satisfied. The execution of the taskiter begins at Step ⓪ of Figure 5.3. The original node (typically Rank 0) creates and offloads a parentless copy of the taskiter to each other rank, and then it executes the taskiter itself. By running the taskiter task, each rank builds a local copy of the full task dependency graph for a single iteration depicted by Step ①. When the taskiter has the *unroll* clause, this "single iteration" may be more than one iteration of the underlying loop.

```
#pragma oss taskiter depend(weakinout : x, y, a, b)
for (int it = 0; it < NUM_ITERATIONS; it++)
{
    // Task 1
    #pragma oss task depend(in : x, y) depend(out : a) node(0)
    {
        ...
    }

    // Task 2
    #pragma oss task depend(in : a, y) depend(out : b) node(1)
    {
        ...
    }

    // Task 3
    #pragma oss task depend(in : a, b) depend(out : x) node(0)
    {
        ...
    }

    // Task 4
    #pragma oss task depend(in : x, b) depend(out : y) node(1)
    {
        ...
    }
}
```

Listing 5.5 Example OmpSs-2@Cluster distributed taskiter implementation of the program of Figure 2.9a, with the mapping from task to rank indicated using the "node" clause.

5.4.3 Partition The Taskiter Graph

Once Rank 0 has finished executing the taskiter task and has created all the sub-tasks, it performs Step (2) of Figure 5.3, which partitions the dependency graph for execution by the processes. Our approach can leverage any partitioning algorithm, and it does not require a fixed or deterministic method, unlike StarPU-MPI [11] and OmpSs@cloudFPGA [22] (see Section 3). The current prototype uses a static partition controlled by the node clause.

Listing 5.5 updates the example program of Figure 2.9a to use OmpSs-2@Cluster with distributed taskiter, and it adds the node clause on each task to indicate the partition that will be used in the rest of this section. Tasks 1 and 3 are executed on Rank 0 and Tasks 2 and 4 are executed on Rank 1, enabling wavefront parallelism across two nodes.

5.4.4 Creating The Local Taskiter Graph

Step ③ of Figure 5.3 translates the full dependency graph into the local Directed Cyclic Task Graph (DCTG) for execution by the current process, and it pre-computes the MPI data transfers that involve the current process. All ranks do this step concurrently, as illustrated in Figure 5.3. There are two sub-steps: (1) insert communication tasks and (2) create the local DCTG.

5.4.4.1 Inserting Communication Tasks

Communication among ranks is done by dedicated tasks to exploit the existing task graph to control the ordering and overlap of communication and computation. A send task has an *in* access over the data to send since the MPI send only needs to read the latest version. A receive task has an *out* access since the MPI receive will update its buffer with the new data version.

The algorithm, implemented in the runtime, to add the send and receive tasks is shown in Figure 5.5. It starts from the *top map*, an existing data structure that maps each region to the first task that accesses it and matches these regions to regions in the *bottom map*, which is also a data structure, however, containing the last task that accessed a specific memory region. The top and bottom maps always use the fragmented regions dependency system as a link between a task's predecessors and subtasks. For the discrete dependency system, it is inherited from the usual SMP taskiter support. When using the region dependency system, an extra pass is required to fragment both maps fully to match the finest access granularity. The complexity of this algorithm is $\mathcal{O}(R)$, where R is the sum over tasks of the number of regions accessed by the task. The value of R corresponds to the overall number of iterations of the `while` loop on line 8 of Figure 5.5. Figure 5.4 shows the example in Listing 5.5 fully fragmented with mapping the top and bottom map data structures and how the runtime perceives it.

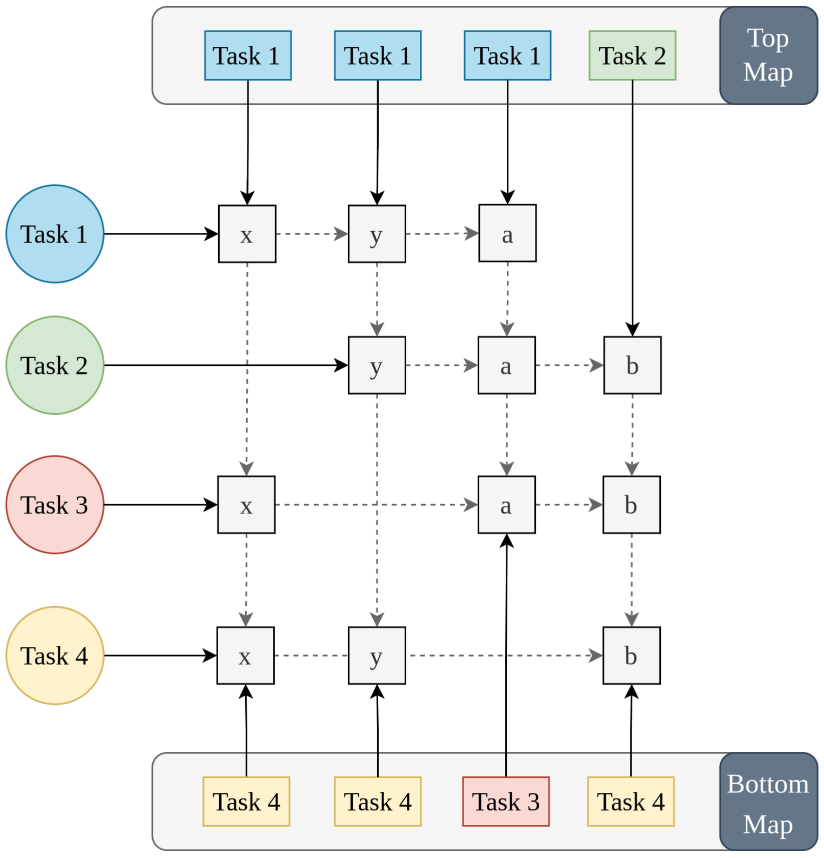


Figure 5.4 Top and Bottom map data structures of the example in Listing 5.5. Solid arrows refer to the first tasks accessing a region. Accesses linked with horizontal dashed arrows are accessed by the same task (e.g., x,y, and a are accessed by Task 1). Vertical dashed arrows indicate the next task accessing a specific region (e.g., the region y is accessed by Task 1, Task 2, and Task 4).

Figure 5.6a shows the output of the algorithm of Figure 5.5 for Rank 0 of the partitioned program in Listing 5.5. The tasks that were created locally by the taskiter parent in Step (1) but will not be executed locally on Rank 0, i.e., Task 2 and Task 4 are greyed out. We assume that the virtual addresses of the variables are in the order a, b, x, y. The loop on line 3 processes each region in the top map in virtual address order, in this case starting with a. Next, the while loop on line 8 considers each task access that contains a, starting with the out access of Task 1. This is the first write to a ($lastWriter$ is none on line 20) so, the empty set of $initialReaders$ is captured on line 21: since the first access is a write, performing any data transfers for the cyclic edges will not be necessary. The next access to the same region is the **in** access of Task 2. This task reads the data, but it is not yet present on Rank 1 ($access.rank \notin readerRanks$ on line 9), so it requires a data transfers from Rank 0, the $lastWriter$, to Rank 1, which executes the task. Since the current rank is Rank 0, a send task is created on line 13. Following the same procedure, Rank 1, creates the matching receive task on line 15. This completes all the accesses to `a`, so the loop on line 3 continues for `b`, which follows a similar process, except that the current rank, Rank 0, needs to create a receive task. The process for `x` is slightly different because the first access to `x` is the in access of Task 1, which reads the value from the as-yet-unknown last writer in the previous iteration. No send–receive pair is created (lines 11 to 15 are skipped), but Rank 0 is added to $readerRanks$ on line 17. Later, once all accesses have been considered, the loop on line 26 will check the $initialReader$, rank 0. Since $lastWriter$ is also rank 0, no send-receive pair is needed. Finally, considering `y`, the first access is the in of Task 0, and the loop on line 26 will also check the $initialReader$, Rank 0. But this time, since the $lastWriter$ is Task 4 on Rank 1, there is no copy of this data on Rank 0, so a receive task for the cyclic read-after-write is created on line 31. This also means that iteration 0 on the current rank will require a valid copy of the data before the taskiter. This is recorded by adding the region to $initialValues$ on line 32.

The algorithm in Figure 5.5 ensures that the iterations of each send and receive task are always serialized, irrespective of the algorithm used to execute the tasks (Section 5.4.5, which serializes the iterations of any particular task in any case). Each receive is serialized due to the write-after-write dependency on its *out* access. Each send is serialized due to the write-after-read dependency from the send task (which has an *in* access) to the $lastWriter$ (defined when the send task is created on line 13) in the next iteration, which has an *out* or *inout* access. MPI sends and receives the same data in different iterations and are therefore posted one at a time and in order. The MPI tags for corresponding sends and receives always match since the sending and receiving ranks follow the same deterministic algorithm on the same task graph. The MPI tag is given by $mpiTag$, which is initialized to zero on line 1 and incremented on lines 16 and 33.

```
$mpiTag \leftarrow 0$                                                    $\triangleright$ Current MPI tag
$initialValues \leftarrow \emptyset$                      $\triangleright$ Data that may need fetching for iteration 0
for all $(region, access) \in topMapAccesses$ do
    $lastWriter \leftarrow none$                          $\triangleright$ Last writer of $region$ by sequential order
    $readerRanks \leftarrow \emptyset$                    $\triangleright$ Ranks with valid copy of latest version of $region$
    $initialReaderRanks \leftarrow \emptyset$             $\triangleright$ Reading ranks of $region$ from prev. iteration
    $\triangleright$ Add send and receive tasks for non-cyclic dependencies $\triangleleft$
    while $access \neq none$ do
        if $access.type \neq OUT$ and $access.rank \notin readerRanks$ then
            $\triangleright$ Task reads data, but it is not yet present locally $\triangleleft$
            if $lastWriter \neq none$ then
                if $currentRank = lastWriter.rank$ then
                    Insert send task of $region$ with tag $mpiTag$ before $access.task$
                else if $currentRank = access.rank$ then
                    Insert receive task of $region$ with tag $mpiTag$ before $access.task$
                $mpiTag \leftarrow mpiTag + 1$
            $readerRanks \leftarrow readerRanks \cup access.rank$
        if $access.type = OUT$ or $access.type = INOUT$ then
            $\triangleright$ Task writes data $\triangleleft$
            if $lastWriter = none$ then
                $initialReaderRanks \leftarrow readerRanks$
            $lastWriter \leftarrow access.task$
            $readerRanks \leftarrow \{currentRank\}$
        $access \leftarrow access.next$
    $\triangleright$ Add send and receive tasks for cyclic dependencies $\triangleleft$
    for all $initialReaderRank \in initialReaderRanks$ do
        if $initialReaderRank \notin readerRanks$ then
            if $currentRank = lastWriter.rank$ then
                Insert send task of $region$ with tag $mpiTag$ at end
            else if $currentRank = initialReaderRank$ then
                Insert receive task of $region$ with tag $mpiTag$ at end
                $initialValues \leftarrow initialValues \cup region$
            $mpiTag \leftarrow mpiTag + 1$
            $readerRanks \leftarrow readerRanks \cup initialReaderRank$
```

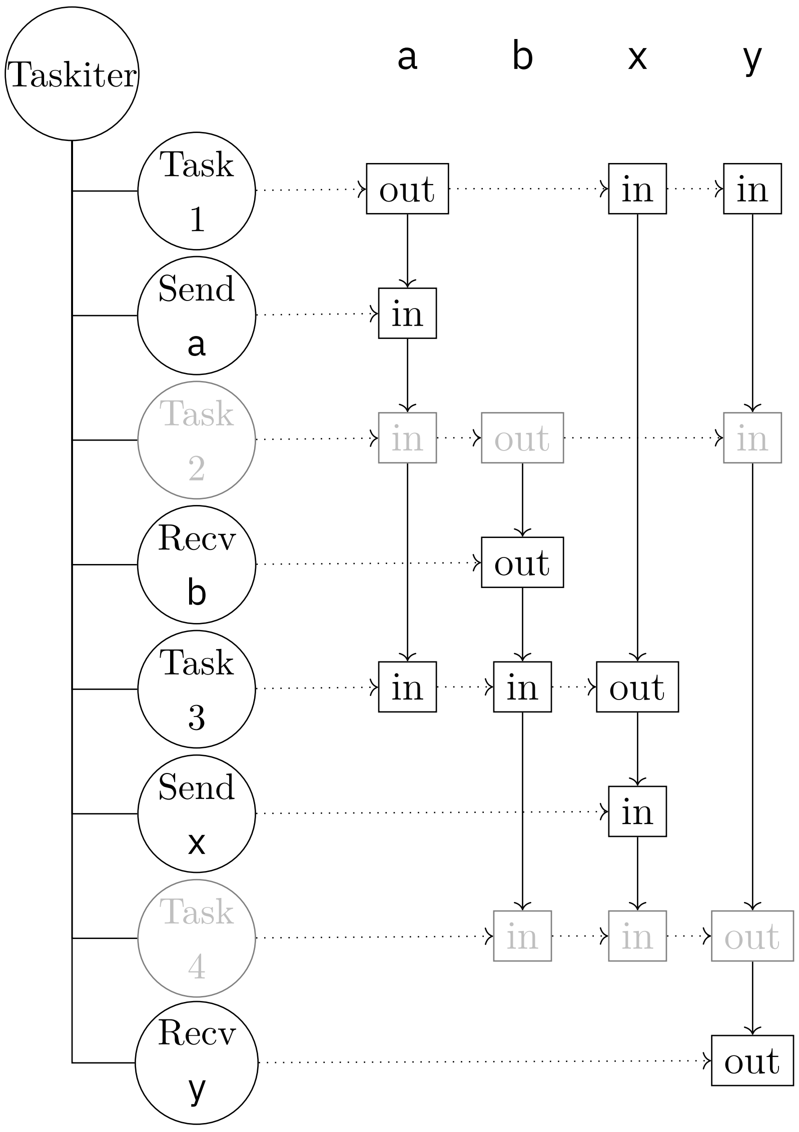
Figure 5.5 Insertion of communication (send and receive) tasks. Communication is done by tasks, ensuring asynchronous execution with maximum communication–computation overlap. This deterministic algorithm runs on all ranks on the same full task graph, so sends and receives on different ranks will always match.

5.4.4.2 Create the local DCTG

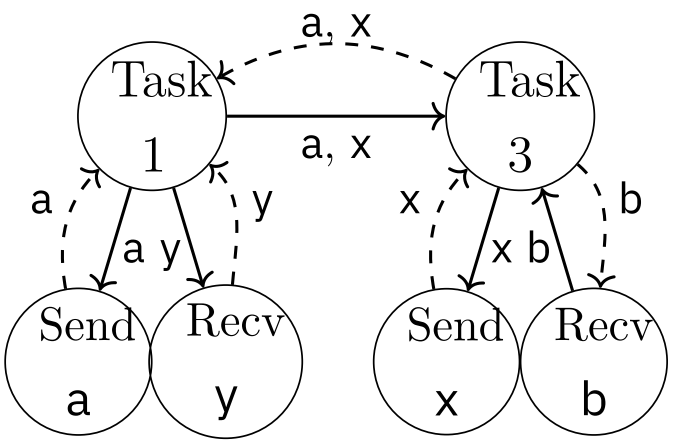
The local dependency graph, a sequential graph on task accesses, is converted into the local Directed Cyclic Task Graph (DCTG). The process is the same as for SMP, and the result for the example program is shown in Figure 5.6b.

5.4.4.3 Fetch input data

As soon as Step ③ has finished on the current node, Step ④ fetches all of the input data the taskiter needs. There is no need for a global barrier between Steps ③ and ④. The algorithm in Figure 5.5 has already determined, in $initialValues$, all the regions whose initial version, before the taskiter, is read by the first iteration of at least one task. These regions will require fetching to this node unless the node already has a copy of the data. The runtime uses its normal data transfer mechanism, which checks whether a data transfer is actually needed, merges contiguous data transfers , and programs the data transfers using non-blocking MPI calls.



(a) Local dependency graph on Rank 0 after inserting the send and receive tasks. The tasks not executed on Rank 0, i.e., Task 2 and Task 4, are disabled and colored in gray.



(b) Local DCTG on Rank 0.

Figure 5.6 Regular dependency graph for a single iteration and directed cyclic task graph for Rank 0 of the example program of Listing 5.5.

5.4.5 Single Iteration Execution

Once the data transfers in Step ④, if any, have been completed on the current node, Step ⑤ proceeds to execute all iterations of the body of the taskiter. It is not usually necessary to have a global barrier between Steps ④ and ⑤, but we add one in our experiments in order to cleanly separate the startup overhead and the time per iteration. This extra barrier has little effect on the total execution time.

The local graph is executed the same way as taskiter on SMP. Communication tasks are ordinary tasks, except that the task body is implemented inside the runtime system rather than the user code. The body of the communication task simply posts the appropriate non-blocking MPI send or receive. The runtime defers the release of the dependencies to the successor tasks, which would otherwise happen immediately until the MPI request completes. Completion of MPI requests is periodically tested by the same dedicated thread that is used for OmpSs-2@Cluster message completion [7].

5.4.5.1 Unroll The Graph

The optional `unroll(n)` clause enables taskiter to support a loop whose task graph repeats every n iterations, especially for loops with a regular dependency graph. An example is a loop which has different graph for odd and even iterations means that the graph will repeat each 2 iterations, hence it would be unrolled with factor of 2.

5.4.5.2 Non-constant Number of Iterations

If the number of iterations is not known before the execution of the loop, then a control task is inserted in a similar way to the approach on SMP as explained in Section 2.3.5.[2] The control task on Rank 0 inherits all the strong accesses of the taskiter, ignoring the weak accesses. The control task on Rank 0 also depends on the previous iteration of the control task on the same rank. The control tasks on all the other ranks only depend on the control task on Rank 0, which is used to copy the value of the condition to all other ranks. Similarly to the SMP implementation, if the condition is false, the control task on each rank cancels the rest of the taskiter. If the taskiter has the unroll clause, then the control task on Rank 0 is studied in the same way as SMP, in order to support the overlapping of tasks from different iterations.

[2]Our prototype implementation does not yet support non-constant iteration counts.

5.4.5.3 Releasing Accesses

Once all local tasks on a remote (non-Rank 0) rank have completed all iterations, then a Task Finished message is sent to Rank 0 in the normal way. Once Rank 0 has completed its own iterations and received notification from all other nodes, it completes the taskiter and releases its accesses. Since Rank 0 knows the partitioning of tasks across ranks (as does every rank), the data locations in the dependency system are updated to correspond to the rank that executes the last writer. If there is no last writer, because the data is only read, then the original location remains the same as before the taskiter

5.5 Methodology and Benchmarks

This part of the thesis evaluated on the MN4 supercomputer and system environment as the one used for evaluating work in Chapter 4 and described in Section 4.5. All benchmarks have a tunable block size, which controls the size of each task, and results are given for the block size that gives the highest performance per iteration.

We test and evaluate the `taskiter` extended approach with five benchmarks of which three, the multi-matvec, multi-matmul, and jacobi adopted and described also in Section 4.5 and summarized in Table 4.2. In addition to two stencil code benchmarks, the heat-gauss, and heat-jacobi. The full list of benchmarks are summarised in Table 5.1.

heat-jacobi is the same 2D heat equation with Jacobi updates. This version has two working arrays and embarrassingly parallel computations inside each timestep to update the array. It is well suited to fork–join parallelism.

heat-gauss is the Gauss–Seidel variant of the 2D heat equation stencil computation discussed in heat-jacobi. It exhibits 2D or 3D wavefront parallelism and has the potential to overlap tasks from multiple iterations, making it a good fit for asynchronous task parallelism.

Table 5.1 Evaluation benchmarks for the `taskiter` approach

Benchmark	**Description**	**Characteristics**	**Parameters**
multi-matvec	Repeated dense matrix–vector multiplication	Small tasks and no communication	Matrix size: 32,768×32,768 elements Number of iterations: 500
multi-matmul	Repeated dense matrix–matrix multiplication	Large tasks and no communication	Matrix size: 32,768×32,768 elements Number of iterations: 6
jacobi	Jacobi iteration [7]	All-to-all communication, good fit for fork–join parallelism	Matrix size: 32,768×32,768 elements Number of iterations: 400
heat-gauss	2D stencil computation with Gauss–Seidel updates	Nearest-neighbor communication, wavefront parallelism suited to asynchronous tasks	Matrix size: 32,768×32,768 elements Number of iterations: 100
heat-jacobi	2D stencil computation with Jacobi updates	Nearest-neighbor communication, good fit for fork–join parallelism	Matrix size: 32,768×32,768 elements Number of iterations: 100

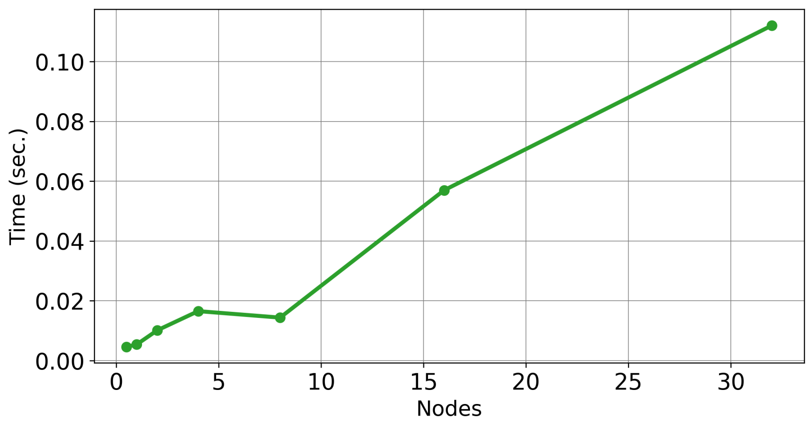
5.6 Evaluation and Results

This section presents the results of the evaluated benchmarks, which are described in Section 5.5 and summarized in Table 5.1.

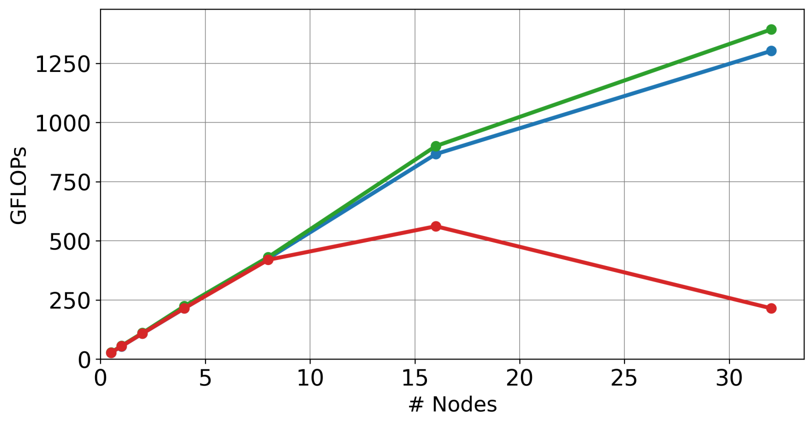
5.6.1 Strong Scalability

In this section, we show the overall results for strong scaling of the five selected benchmarks described in Section 4.5.2 and 5.5 with two processes per node (one per socket). We show the initial overhead and the performance per iteration, excluding the initial overhead. In all plots, the x-axis is the number of nodes, always with two MPI processes per node, i.e., one process per socket. Chart (a) gives the initial overhead (seconds on the y-axis), which is independent of the number of iterations, and the chart (b) gives the performance per iteration for the body of the loop (GFLOPS/s on the y-axis). The colours distinguish the four programming models: blue for fork–join MPI + OpenMP, brown for TAMPI + OmpSs-2, red for the original OmpSs-2@Cluster implementation, and green for the distributed taskiter. All points use the block size that gives the best performance per iteration for the loop's body, for that benchmark and implementation. All data points include error bars, but the error bars are too small to see in almost all cases.

Figure 5.7a shows the overhead for multi-matvec, which has a maximum value of just 0.11 seconds on 32 nodes. This overhead corresponds to Steps ① to ④ and Step ⑥ in Section 5.4, and for this benchmark, it scales roughly linearly with the number of nodes since the optimal block size corresponds to a small number of tasks per core, and all tasks always have the same number of accesses. Looking at Figure 5.7b, we see that, after paying this small cost, the distributed taskiter variant achieves similar scaling behaviour to the baseline fork–join MPI version. There is a slight improvement over fork–join MPI by 7.0% on 32 nodes, likely because of minor differences between the OpenMP and Nanos6@Cluster runtimes. This result is a large improvement compared with the original OmpSs-2@Cluster implementation, which scales to just 8 nodes and is 6 times slower than fork–join MPI on 32 nodes. This benchmark has no inter-node communication, so the poor scaling of the OmpSs-2@Cluster implementation is due to the control message overhead for task offloading and dependency management, which is entirely eliminated by the distributed taskiter approach. Since there is no inter-node communication, the asynchronous TAMPI + OmpSs-2 results have been omitted.



(a) Initial overhead

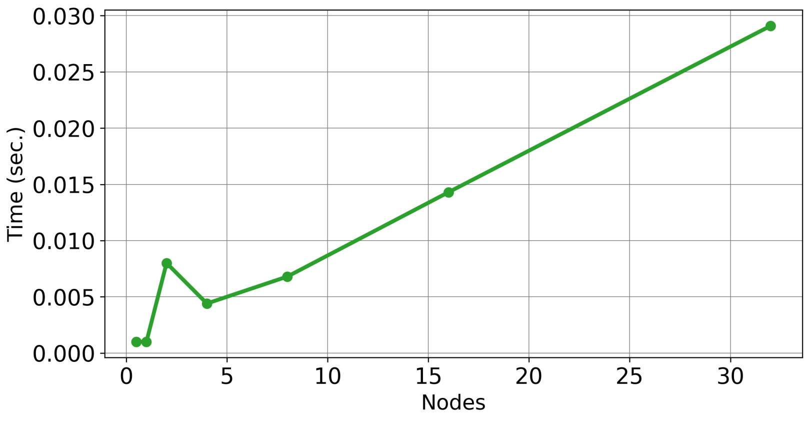


(b) Performance per iteration

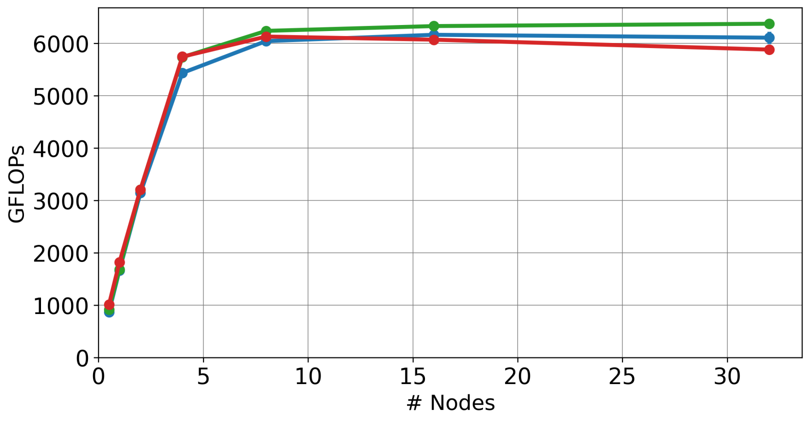
🟦 Fork–join MPI+OpenMP 🟥 OmpSs-2@Cluster 🟩 Distributed Taskiter

Figure 5.7 Benchmark: multi-matvec

Figure 5.8a shows the initial overhead for multi-matmul, which has a maximum value of just 0.029 seconds on 32 nodes. Again, the overhead scales roughly linearly with the number of nodes. As seen in Figure 5.8b, all three variants scale similarly for this benchmark. The drop in scalability beyond 4 nodes is due to unusual behaviour from the Math Kernel Library (MKL) library, which achieves approximately 3× higher throughput for block sizes of 256 elements or more. For a 32768 × 32768 matrix, the optimal block size does not fully use the available compute resources when there are 8 or more nodes.



(a) Initial overhead



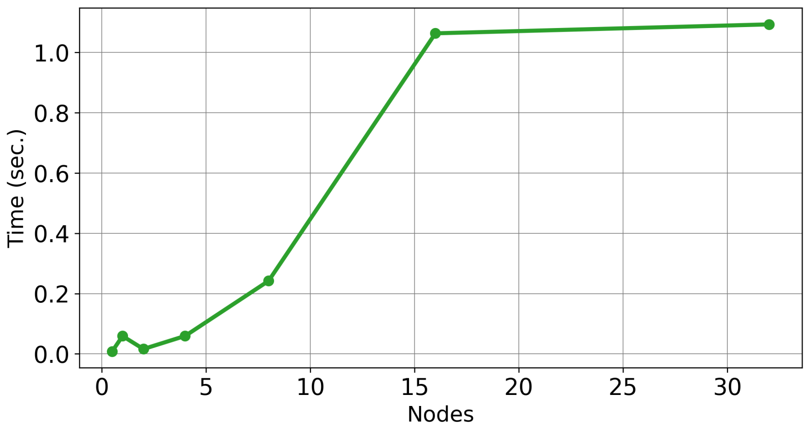
(b) Performance per iteration

🟦 Fork–join MPI+OpenMP 🟥 OmpSs-2@Cluster 🟩 Distributed Taskiter

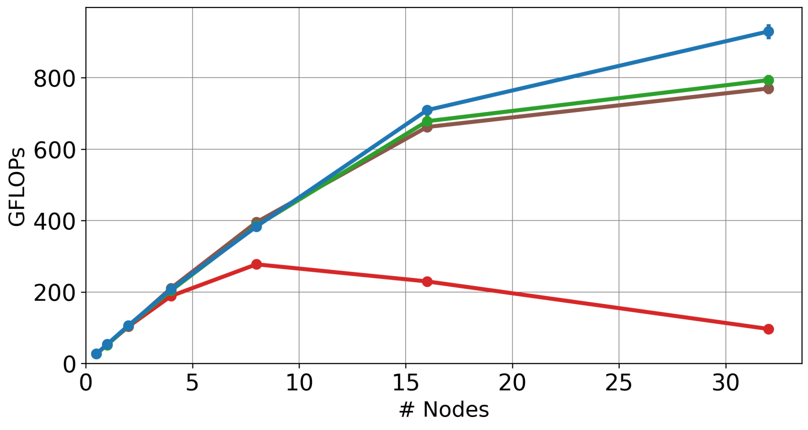
Figure 5.8 Benchmark: multi-matmul

Figure 5.9a shows that the initial overhead for jacobi grows roughly quadratically from 1 to 16 nodes. This is due to the benchmark's all-to-all communication, which means that the number of tasks and the number of accesses per task both grow roughly linearly in the number of cores, which combine to cause quadratic growth. Beyond 16 nodes, it is no longer beneficial to subdivide the tasks. The maximum overhead on 32 nodes is 1.1 seconds. In Figure 5.9b, we see that the distributed taskiter approach matches the fork–join MPI version, within 4.4% on up to 16 nodes, but it drops to 15% below the fork–join MPI version on 32 nodes. There are two reasons for this. Firstly, the fork–join MPI version uses collective communication, whereas the distributed taskiter uses point-to-point communication. Secondly, the distributed taskiter has asynchronous communication inside the tasks instead of fork–join parallelism. We see that the results closely match, within 3%, those for the asynchronous TAMPI +

OmpSs-2 version on up to 32 nodes, which also has point-to-point communication inside tasks. Future work may investigate ways to use collective communication or merge communication into existing tasks. In any case, the results already greatly outperform the original OmpSs-2@Cluster, which is 9.6 times slower than fork–join MPI + OpenMP,



(a) Initial overhead



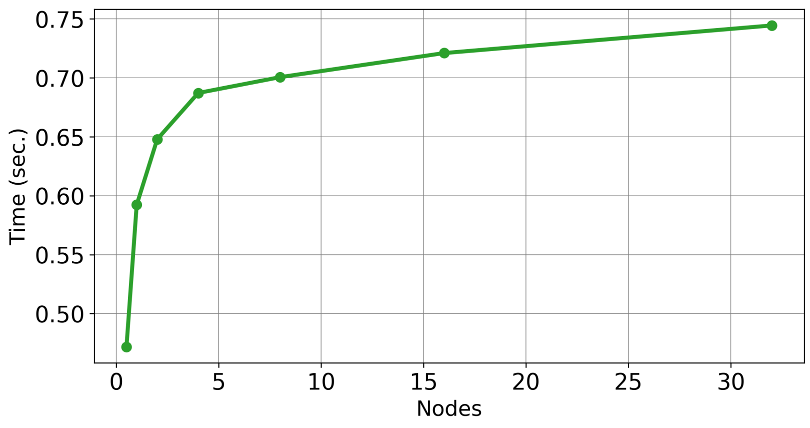
(b) Performance per iteration

🟦 Fork–join MPI+OpenMP 🟫 Asynchronous TAMPI+OmpSs-2
🟥 OmpSs-2@Cluster 🟩 Distributed Taskiter

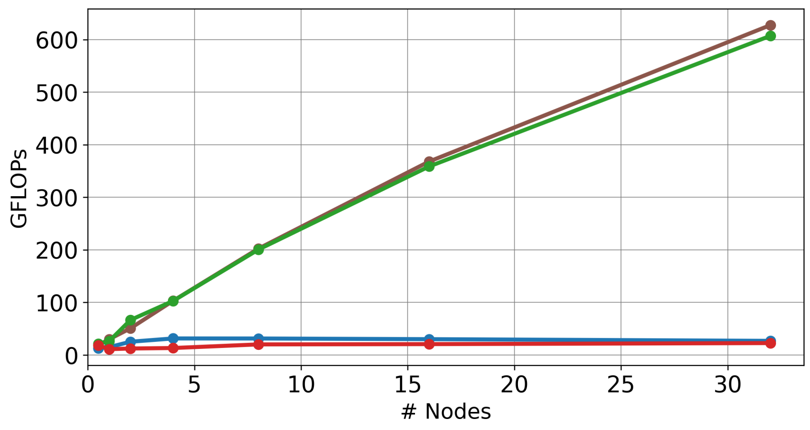
Figure 5.9 Benchmark: jacobi

Figure 5.10a shows the overhead for heat-gauss. Because of the 3D wavefront parallelism is the most efficient method of using the smallest block size with acceptable performance overheads. The overhead is, therefore, roughly constant, rising from 0.47 seconds on 1 node to 0.74 seconds on 32 nodes. Figure 5.10b shows that the fork–join MPI + OpenMP version has poor performance, limited by the 2D wavefront parallelism inside each timestep. By enabling 3D wavefront parallelism, the asynchronous TAMPI

+ OmpSs-2 version achieves much higher performance, reaching performance 11.4 times faster than fork–join MPI + OpenMP on 32 nodes. The distributed taskiter version achieves similar performance, at 11.0 times faster than fork–join MPI + OpenMP on 32 nodes. In contrast, the OmpSs-2@Cluster implementation performs even worse than fork–join MPI + OpenMP, being 2.4 times *slower*, on 32 nodes.



(a) Initial overhead

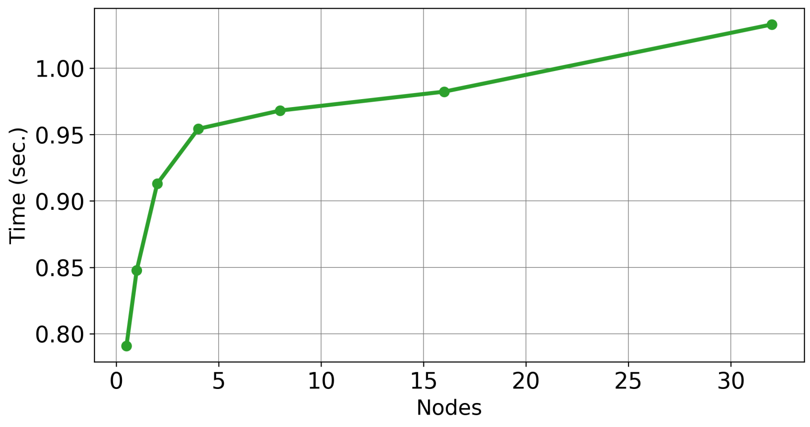


(b) Performance per iteration

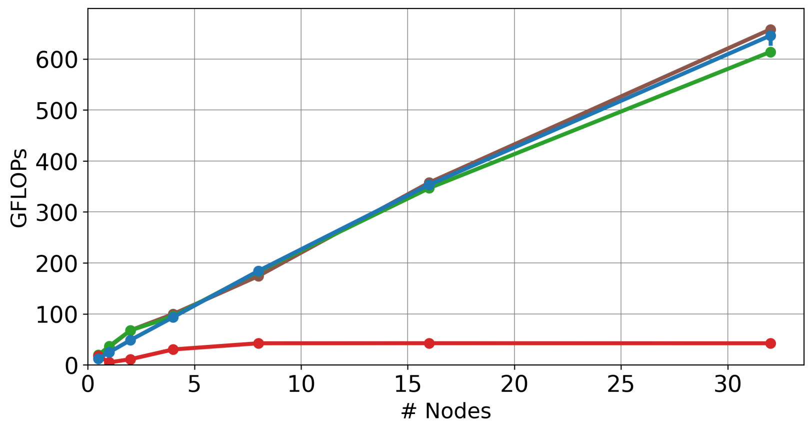
🟦 Fork–join MPI+OpenMP 🟫 Asynchronous TAMPI+OmpSs-2
🟥 OmpSs-2@Cluster 🟩 Distributed Taskiter

Figure 5.10 Benchmark: heat-gauss

Figure 5.11a shows the overhead for heat-jacobi, the Jacobi version of the 2D heat-equation stencil computation. Each iteration is embarrassingly parallel, and the optimal block size is roughly constant from 1 to 32 nodes. The overhead is almost constant, rising to a maximum of just over 1.0 seconds on 32 nodes. Finally, in Figure 5.11b, all versions except the original OmpSs-2@Cluster implementation achieves similar scaling to at least 32 nodes. The distributed taskiter version is within 5.0% of the performance per iteration of fork–join MPI, which is a dramatic improvement in comparison with the original OmpSs-2@Cluster implementation, which is 15 times slower than fork–join MPI.



(a) Initial overhead



(b) Performance per iteration

🟦 Fork–join MPI+OpenMP 🟫 Asynchronous TAMPI+OmpSs-2
🟥 OmpSs-2@Cluster 🟩 Distributed Taskiter

Figure 5.11 Benchmark: heat-jacobi

Overall, these results show that the initial overheads of distributed taskiter are acceptable, with a maximum of 1.1 seconds. The majority of this time is in the Step ③ graph translation described in Section 5.4.4, which is done by our implementation on one thread but could easily be parallelized across all 24 threads in each process. After paying that small cost, the performance-per-iteration matches or exceeds fork–join MPI + OpenMP, with a slight drop of 15% only for the jacobi benchmark on 32 nodes. For heat-gauss, which benefits from asynchronous communication, the performance is similar to asynchronous TAMPI + OmpSs-2, at 11.0 times faster than fork–join MPI +

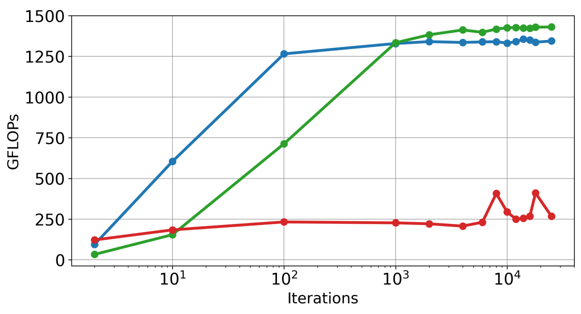
OpenMP. In four out of the five benchmarks, the performance-per-iteration far exceeds that of the original OmpSs-2@Cluster, which is up to 15 times slower than fork–join MPI + OpenMP.

5.6.2 Iterations Count Performance

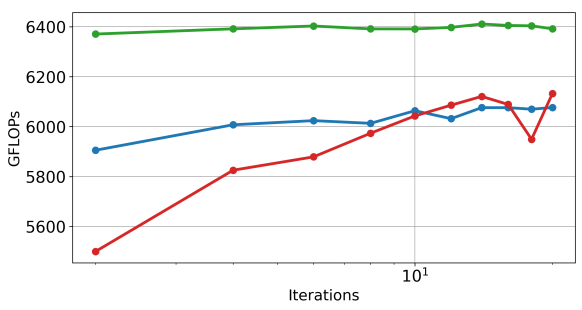
In this section, we show the overall performance, including all overheads, as a function of the number of loop iterations executing on 32 nodes (64 processes). The x-axis is the number of iterations on a log scale. The y-axis is the overall performance in GFLOPS/s, which, unlike the results shown in Section 5.6.1, includes the initial overhead. These results were observed afresh, not estimated synthetically by combining *subfigure-a* and *subfigure-b* for each of the benchmarks in Figures from 5.7 to Figure 5.11.

We see that in Figure 5.12, including the startup overhead, the distributed taskiter implementation achieves higher performance for multi-matmul, heat-gauss and heat-jacobi then the original OmpSs-2@Cluster version from the first iteration (unrolled by two). In many cases, few iterations are required to achieve performance close to fork–join MPI + OpenMP. For multi-matvec, it exceeds the original version from 10–100 and 100–1000 iterations, respectively. The distributed taskiter version has slightly higher performance than fork–join MPI + OpenMP, from 100 or more iterations for multi-matvec and from the first iteration for multi-matmul (note the zoomed y-axis for multi-matmul, in order to expose the small differences between versions). For jacobi, distributed taskiter's performance steadily increases up to about 15% below that of fork–join MPI + OpenMP. For heat-gauss, the two versions that allow an asynchronous overlap of timesteps, i.e., the TAMPI + OmpSs-2 and distributed taskiter versions, have performance growing over the first approximately 1000 iterations, due to the increasing ability to overlap tasks from different timesteps through 3D wavefront parallelism. Finally, for heat-jacobi distributed taskiter reaches close to the performance of MPI + OpenMP after about 1000 iterations.

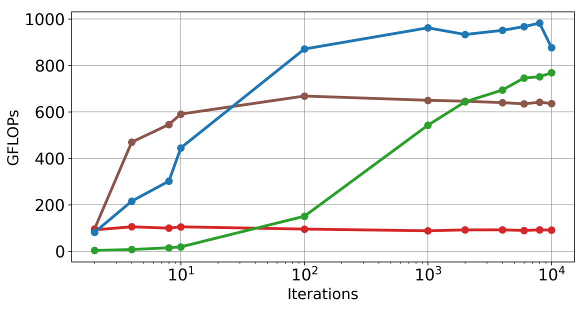
It is important to note that the startup overhead has not been optimized in our current distributed taskiter implementation. As remarked above, most of the overhead is in the Step ③ graph translation described in Section 5.4.4. This is currently done by a single thread, but it could be parallelized across all 24 threads in each process.



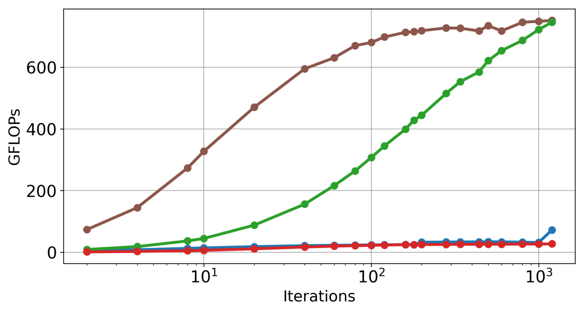
(a) multi-matvec on 32 nodes



(b) multi-matmul on 32 nodes

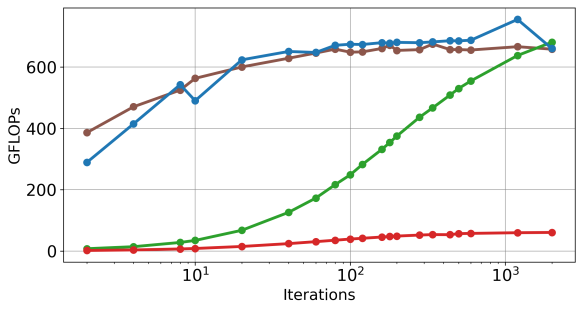


(c) jacobi on 32 nodes



(d) heat-gauss on 32 nodes

🟦 Fork–join MPI+OpenMP
🟫 Asynchronous TAMPI+OmpSs-2
🟥 OmpSs-2@Cluster
🟩 Distributed Taskiter



(e) heat-jacobi on 32 nodes

Figure 5.12 Overall performance, including all overheads, as a function of the number of iterations. We did not provide TAMPI+OmpSs-2 for multi-matvec and multi-matmul as they have no communications that would benefit from the TAMPI asynchronous model.

5.7 Initial Overhead Analysis

Figure 5.13 plots the initial overhead in seconds on the y-axis versus the number of task access regions on the x-axis emphasized by individual benchmarks. The number of access regions is defined in Section 5.4.4.1 as the sum over tasks of the number of regions accessed by the task, where it is denoted R. Data points are shown for all benchmarks, numbers of nodes and a sweep of block sizes (not only the optimal block size used in Section 5.6.1). There is a small effect from the benchmark, particularly for jacobi, which features all-to-all communication and more than average communication per access. Nevertheless, we see a strong linear relationship between the number of accesses and the initial overhead across four orders of magnitude, with just a few outliers, justifying the theoretical $O(R)$ complexity given in Section 5.4.4.1.

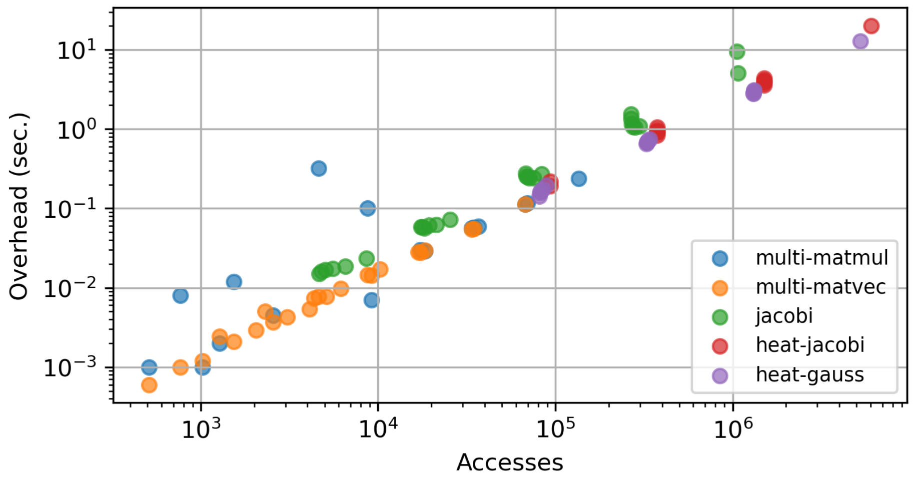


Figure 5.13 Initial overhead as a function of the number of accesses.

Figure 5.14 shows how the number of accesses, in turn, depends on the block size. The x-axis is the block size, and the y-axis is the number of accesses, when executed on a single node. For multi-matvec and multi-matmul, which appear on top of each other since they have the same number of accesses, the number of accesses is a constant times the number of tasks, which is inversely proportional to the block size. The other benchmarks have their number of accesses inversely proportional to the square of the block size. For jacobi, this is because the number of tasks and accesses per task is inversely proportional to the block size. For heat-gauss and heat-jacobi, the number of accesses per task is fixed, but the number of tasks is inversely proportional to the square of the block size.

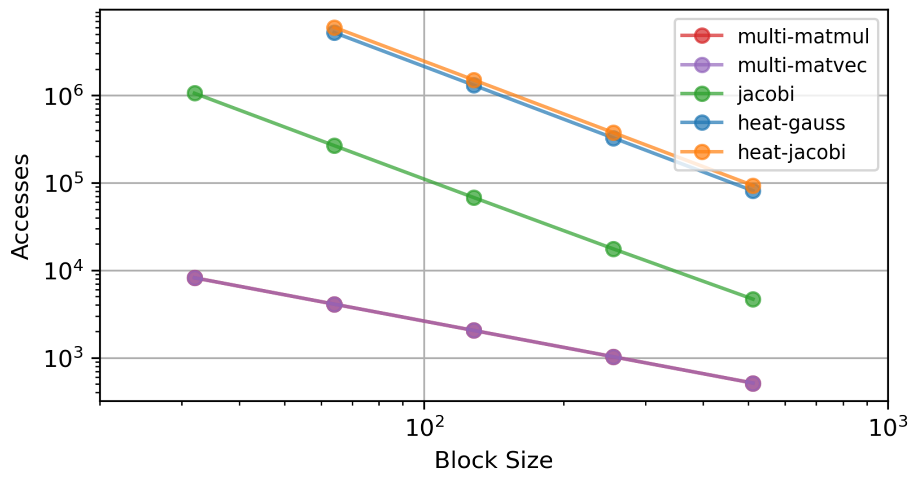


Figure 5.14 Number of accesses depending on block size.

5.8 Quantifying Productivity

Table 5.2 attempts to quantify the productivity of MPI+OpenMP, TAMPI+OmpSs-2 and distributed taskiter, using the commonly-used metric of lines of code. Quantifying the productivity of a programming system is a well-known problem, as indicated in Section 4.6.4, and lines of code do not account for complexity, ease of understanding or the effects of varying programmer skill levels and preferences. It is, however, the best available metric because it is easily measured and understood.

The kernel code is isolated in a separate source file and is the same for all implementations under study. The control code counts all source lines in the main loop of the benchmark, which makes a function call to the kernel code. [3] It excludes argument processing, memory allocation, initialization, and verification of the result. For multi-matvec and multi-matmul, which involve no communication among nodes, there is essentially no difference between the three approaches. For jacobi, the distributed taskiter version has a similar number of lines of code to MPI, which only requires a collective MPI_Allreduce, while the TAMPI+OmpSs-2 more than triples the number of lines of code (+210.7%). For heat-gauss and heat-jacobi, the asynchronous TAMPI+OmpSs-2 version increases the number of lines by at least a third, while the distributed taskiter version has just over half as many lines of code.

[3] There are some small differences between the numbers of lines of code in Table 5.2 and code present in listings 5.2 to 5.4, as the latter is reformatted for conciseness and clarity.

Benchmark	Kernel				
	Fork–join MPI+OpenMP	TAMPI+ OmpSs-2		Distributed taskiter	
multi-matvec	11	11	(+0%)	11	(+0%)
multi-matmul	18	18	(+0%)	18	(+0%)
jacobi	21	21	(+0%)	21	(+0%)
heat-gauss	40	40	(+0%)	40	(+0%)
heat-jacobi	40	40	(+0%)	40	(+0%)

Table 5.2 Number of lines of code for the kernels of of the benchmarks.

Benchmark	Control				
	Fork–join MPI+OpenMP	TAMPI+ OmpSs-2		Distributed taskiter	
multi-matvec	15	15	(+0%)	16	(+6.67%)
multi-matmul	15	15	(+0%)	16	(+6.67%)
jacobi	28	87	(+210.71%)	27	(-3.57%)
heat-gauss	45	60	(+33.33%)	25	(-44.44%)
heat-jacobi	54	76	(+40.74%)	30	(-44.44%)

Table 5.3 Number of lines of code for the control code of the benchmarks.

5.9 Conclusion

Despite being very productive, distributed Sequential Task Flow (STF) models suffer from limited performance and scalability for fine- and medium-grained tasks. In this work, we presented an extension to OmpSs-2@Cluster that addresses this issue for applications with common iterative patterns while maintaining the productive OmpSs-2@Cluster programming model. While the existing OmpSs-2@Cluster implementation scales to only about 4 or 8 nodes with medium-scale tasks, our approach scales to at least 32 nodes, with a maximum slowdown of 15%, compared with fork–join MPI + OpenMP. When the application has the potential to overlap iterations, for example, the 2D heat equation stencil calculation with Gauss–Seidel updates, our approach discovers significantly more parallelism than fork–join MPI + OpenMP. This results in up to 11.0 times higher performance on 32 nodes, which is on-a-par with state-of-the-art asynchronous TAMPI + OmpSs-2. As such, the model combines the productivity of STF models with the performance of state-of-the-art MPI+X approaches by exploiting the iterative nature of scientific applications. It also avoids the synchronization and deadlock issues of an MPI+X approach.

CHAPTER 6

Conclusions and Future work

Despite being very productive, distributed Sequential Task Flow (STF) models, such as OmpSs-2@Cluster, suffer from limited performance and scalability for fine- and medium-grained tasks. This limitation stems from the sequential bottleneck imposed by the construction of the task dependency graph.

In this thesis, we proposed two approaches to improve the efficiency and scalability of STF models. The first approach employs task nesting to construct subgraphs of the dependency graph concurrently. Task nesting is unfortunately rarely used in production codes, and we identify one issue that increases complexity: tasks can only be constructed once the addresses and sizes of all its accesses, which must cover all descendent task accesses, are known. We solve this issue by introducing the `auto` access type. An `auto` access can be used to support tasks that allocate and return new memory regions and tasks whose subtasks access memory regions defined by previous tasks. It also provides an incremental path to task nesting by allowing the runtime to infer the task accesses on behalf of subtasks. Additionally, we introduced the `none` access type to complement `auto`, allowing for finer control over dependency information and enhancing concurrency between subtasks. We offer a straightforward implementation and optimizations, while maintaining the existing unified method of specifying task information to the runtime without begin obscure (e.g. using sentinels), ensuring clarity and continuity in task creation without introducing barriers (e.g. using a `taskwait`) or interruptions.

We evaluate our approach firstly for dependencies without nested tasks. On SMP, we used a hypermatrix multiplication followed by a Cholesky decomposition benchmark, which shows the `auto` potential compared `taskwait` implementation with 5% performance increase limited by the final part of the Cholesky decomposition that might need an improved scheduler. On clusters, we evaluated the strong scalability of Barnes–Hut

n-body application on up to 32 nodes, which achieves 1.4 times speedup on 32, and a maximum of 195K particles per second on 16 nodes compared to 137K for the manual `taskwait` code. Secondly, we evaluate task nesting on SMP for `matmul-smp` and `n-body-smp` applications reaching 19.6% and 10%, respectively, of the original manual version. Lastly, we evaluate nesting on clusters up to 32 nodes with four benchmarks: `multi-matvec`, `multi-matmul`, `jacobi`, and `cholesky`. We were fairly within 4% of the original nesting with weak tasks of results in Aguilar et al. [7] for 1 to 16 nodes and extend to 32 nodes.

Our second approach exploits the nature of many HPC applications that use iterative methods or multi-step simulations. These applications create the same task dependency graph on each iteration. We take advantage of this information to execute the loop body once and convert the task dependency graph into a DCTG, which is reused during executing the remaining iterations of the loop. We define the programming model based on the `taskiter` construct proposed by Álvarez and Beltran [1]. Knowing the fact that the DCTG will be the same during the entire execution, we store it in a simple representation without locking or complex lock-free data structures. The previous fact also reduces the impact on the execution time of the more powerful but expensive fragmented regions dependency (see Sections 2.1.2) system [36], since dependency system operations are only performed for the tasks in a single iteration and once at the DCTG creation time.

The distributed `taskiter` approach scales up to a 32 nodes, compared to about 4 or 8 nodes with the existing original OmpSs-2@Cluster implementation in [7] when evaluated for the same benchmarks. It shows a maximum slowdown of 15%, compared with fork–join MPI + OpenMP. For cases with a higher potential of iterations overlapping, the distributed `taskiter` achieved up to 11.0 times higher compared to the fork–join MPI + OpenMP reaching asymptotic performance with state-of-the-art asynchronous TAMPI + OmpSs-2.

Both approaches were seamlessly integrated within the Nanos6 runtime implementation of the OmpSs-2@Cluster model, requiring minimal adjustments to the compiler. Incorporating the `auto`, and `none` access types, which resembled the implementation of the existing types shown in Table 2.1. Similarly, the `taskiter` construct was added to the compiler with no effort, which matches the existing `taskloop` construct.

In conclusion, this thesis presented solutions to the sequential bottleneck inherent in the OmpSs-2@Cluster distributed task programming model without compromising the model's interface, simplicity, or productivity. We facilitate the development of applications with nested tasks and provide a simpler approach for users to accelerate the initial development of their code by annotating dependencies that align with the underlying problem's functionality. Moreover, the utilization of common iterative patterns underscores the potential for further scalability.

6.1 Future Work

This thesis opens several avenues of future work, the most important of which are described below.

6.1.1 auto and none clauses

The current implementation of the `auto` and `none` clauses is supported in the *fragmented-regions* dependency system (see Section 2.1.2), which allows partial or complete overlapping between dependency memory regions. Future work is to support `auto` and `none` in the *discrete* dependency system so that it is compatible with both OmpSs-2 and OpenMP models.

6.1.2 Combining auto and distributed taskiter

The two main contributions of this thesis, automatic data access aggregation (Chapter 4) and distributed taskiter (Chapter 5), are separate contributions targeting different types of applications, with non-iterative and iterative behaviour, respectively. The current runtime implementation allows the use of both contributions separately inside the same application, for example, where part of the program has an iterative structure and another part of the program does not. This is because both parts of this thesis have been implemented inside the same branch of the Nanos6@Cluster runtime system (data access aggregation has already been merged onto the main development branch).

It is, however, possible to integrate these contributions in two ways. Firstly, the `auto` keyword can be used to infer all the accesses of the distributed taskiter itself. As described in Section 5.3, the distributed taskiter must have a full specification of its accesses, covering the accesses of all top-level tasks in at least the weak variant. This is a clear use case for automatic inference of weak accesses using the `auto` keyword. The only accesses that would still need to be specified explicitly by the programmer are the strong accesses made outside subtasks by the loop body or loop condition. Implementation is expected to be straightforward, but it has not been tested or evaluated yet.

Secondly, the top-level tasks inside a taskiter may create subtasks, just like any other task. These top-level tasks may also, in principle, use the `auto` keyword to infer their weak accesses. So far, however, we have not seen the need to combine taskiter and task nesting in the same loop since the main motivation for task nesting is to address the sequential overhead, which is already amortized across all loop iterations by distributed taskiter. Moreover, expressing the whole dependency graph at the top level gives a fine task granularity, so the partitioner has the most freedom to balance the load across the available resources. A hybrid approach that employs both taskiter and nesting would only be useful at extreme levels of scaling.

One method to integrate `auto` and distributed taskiter in the second is to extend the programming model to declare that each iteration of a top-level task with the `auto` keyword generates subtasks that cover the same memory regions. Once the runtime has executed a single iteration of all top-level tasks, it could infer the accesses of all top-level tasks using the `auto` keyword and then use the assumption to infer the accesses of all iterations of all top-level tasks. It may also be necessary to extend the definition of communication tasks and the manner of task execution to allow early release (see Section 2.2).

6.1.3 Overhead Optimisations

Both `auto` and `taskiter` can benefit from compiler optimisation and static analysis, such as discovering dependencies that can help with the overhead of the runtime dependency filtering, which can be very expensive. In addition, most of the startup overhead of distributed taskiter can be parallelized across multiple threads and potentially multiple nodes. This includes the creation of the local Directed Cyclic Task Graph (DCTG), insertion of communication and the merging of communications. Such optimisations may be necessary when scaling to large numbers of cores.

6.1.4 Collective Communications

The current implementation uses point-to-point communications, which in some cases reported lower performance compared to that of similar solutions using collectives (see Section 5.6.1). Hence, the substitution of collective communications, where possible, in the place of point-to-point communications is one of the main topics to be investigated in future.

6.1.5 Dynamic Scheduling and Load Balancing

As remarked in Section 5.4.3, our method is compatible with any graph partitioning algorithm, but we have not yet integrated a graph partitioner or performed an evaluation of different partitioning algorithms for our use case. The data affinity hints from the user (see Section 5.2), data transfer costs from the access annotations and estimated task costs from the Nanos6 monitoring interface [93] are all available to the partitioning algorithm. An interesting avenue of future work would be to recognise an unexpected load imbalance and respond by dynamically repartitioning the work in an efficient way.

6.1.6 Non-constant Iterations

Although it might be straightforward, using variable iterations count with `taskiter` requires adding extra logic and communication messages between nodes to identify when the loop ends.

6.1.7 Loop Unrolling

So far, we unroll the loop with a factor of 2 to build the graph; however, by dynamically adjusting the unrolling factor, it's possible to optimise task granularity and exploit available parallelism more effectively.

6.1.8 Summary

In summary, this thesis has identified and built two complementary techniques to improve the scalability of distributed STF tasking models. We hope that future research will build on these foundations along the above lines to create robust and productive distributing tasking models that also deliver high performance and scalability.

References

[1] D. Álvarez and V. Beltran, “Optimizing iterative data-flow scientific applications using directed cyclic graphs,” *IEEE access*, 2023. [Online]. Available: https://doi.org/10.1109/ACCESS.2023.3269902

[2] OpenMP Architecture Review Board, “OpenMP Application Programming Interface, Version 5.2,” 11 2021, accessed: 2022-04-19. [Online]. Available: https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5-2.pdf

[3] T. Kadosh, N. Hasabnis, T. Mattson, Y. Pinter, and G. Oren, “Quantifying openmp: Statistical insights into usage and adoption,” in *2023 IEEE High Performance Extreme Computing Conference (HPEC).* IEEE, 2023, pp. 1–7.

[4] O. Korakitis, S. G. De Gonzalo, N. Guidotti, J. a. P. Barreto, J. C. Monteiro, and A. J. Peña, “Towards OmpSs-2 and OpenACC interoperation,” in *Proceedings of the 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP ’22. New York, NY, USA: Association for Computing Machinery, 2022, p. 433–434. [Online]. Available: https://doi.org/10.1145/3503221.3508401

[5] R. Ferrer, J. Planas, P. Bellens, A. Duran, M. Gonzalez, X. Martorell, R. M. Badia, E. Ayguadé, and J. Labarta, “Optimizing the exploitation of multicore processors and GPUs with OpenMP and OpenCL,” in *International Workshop on Languages and Compilers for Parallel Computing.* Springer, 2010, pp. 215–229.

[6] J. Bosch, X. Tan, A. Filgueras, M. Vidal, M. Mateu, D. Jiménez-González, C. Álvarez, X. Martorell, E. Ayguadé, and J. Labarta, “Application acceleration on FPGAs with OmpSs@FPGA,” in *2018 International Conference on Field-Programmable Technology (FPT)*, 2018, pp. 70–77.

[7] J. Aguilar Mena, O. Shaaban, V. Beltran, P. Carpenter, E. Ayguadé, and J. Labarta, “OmpSs-2@Cluster: Distributed memory execution of nested

OpenMP-style tasks," in *European Conference on Parallel Processing: Euro-Par 2022*, 2022. [Online]. Available: https://doi.org/10.1007/978-3-031-12597-3_20

[8] E. Ayguadé, N. Copty, A. Duran, J. Hoeflinger, Y. Lin, F. Massaioli, X. Teruel, P. Unnikrishnan, and G. Zhang, "The design of OpenMP tasks," *IEEE Transactions on Parallel and Distributed Systems*, vol. 20, no. 3, pp. 404–418, 2008.

[9] OpenMP Architecture Review Board, "OpenMP Application Program Interface version 5.1," Nov. 2020. [Online]. Available: http://www.openmp.org/mp-documents/spec30.pdf

[10] J. M. Perez, V. Beltran, J. Labarta, and E. Ayguadé, "Improving the integration of task nesting and dependencies in OpenMP," in *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2017, pp. 809–818. [Online]. Available: https://doi.org/10.1109/IPDPS.2017.69

[11] C. Augonnet, O. Aumage, N. Furmento, R. Namyst, and S. Thibault, "StarPU-MPI: Task programming over clusters of machines enhanced with accelerators," in *European MPI Users' Group Meeting*. Springer Berlin Heidelberg, 2012, pp. 298–299. [Online]. Available: https://doi.org/10.1007/978-3-642-33518-1_40

[12] R. Hoque, T. Herault, G. Bosilca, and J. Dongarra, "Dynamic task discovery in PaRSEC: a data-flow task-based runtime," in *Proceedings of the 8th Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems*, 11 2017, pp. 1–8. [Online]. Available: https://doi.org/10.1145/3148226.3148233

[13] H. Yviquel, M. Pereira, E. Francesquini, G. Valarini, G. Leite, P. Rosso, R. Ceccato, C. Cusihualpa, V. Dias, S. Rigo, A. Souza, and G. Araujo, "The OpenMP Cluster programming model," in *Workshop Proceedings of the 51st International Conference on Parallel Processing*, ser. ICPP Workshops '22. New York, NY, USA: Association for Computing Machinery, 2023. [Online]. Available: https://doi.org/10.1145/3547276.3548444

[14] J. Klinkenberg, P. Samfass, M. Bader, C. Terboven, and M. Müller, "CHAMELEON: Reactive load balancing for hybrid MPI+OpenMP task-parallel applications," *Journal of Parallel and Distributed Computing*, vol. 138, 12 2019. [Online]. Available: https://doi.org/10.1016/j.jpdc.2019.12.005

[15] K. Sala, X. Teruel, J. M. Perez, A. J. Peña, V. Beltran, and J. Labarta, "Integrating blocking and non-blocking MPI primitives with task-based

programming models," *Parallel Computing*, vol. 85, pp. 153–166, 2019. [Online]. Available: https://doi.org/10.1016/j.parco.2018.12.008

[16] K. Sala, S. Macià, and V. Beltran, "Combining one-sided communications with task-based programming models," in *2021 IEEE International Conference on Cluster Computing (CLUSTER)*, 2021, pp. 528–541. [Online]. Available: https://doi.org/10.1109/Cluster48925.2021.00024

[17] L. Smith and M. Bull, "Development of mixed mode MPI / OpenMP applications," *Sci. Program.*, vol. 9, no. 2,3, p. 83–98, aug 2001.

[18] J. Aguilar Mena, O. Shaaban, V. Lopez, M. Garcia, P. Carpenter, E. Ayguadé, and J. Labarta, "Transparent load balancing of MPI programs using OmpSs-2@Cluster and DLB," in *51st International Conference on Parallel Processing (ICPP)*, 2022. [Online]. Available: https://doi.org/10.1145/3545008.3545045

[19] J. A. Mena, "Methodology for malleable applications on distributed memory systems," Ph.D. dissertation, Universitat Politècnica de Catalunya, 2022. [Online]. Available: http://dx.doi.org/10.5821/dissertation-2117-380814

[20] M. Cosnard and M. Loi, "Automatic task graph generation techniques," in *Proceedings of the Twenty-Eighth Annual Hawaii International Conference on System Sciences*, vol. 2, 1995, pp. 113–122 vol.2.

[21] Barcelona Supercomputing Center. (2021) OmpSs-2 specification. [Online]. Available: https://pm.bsc.es/ftp/ompss-2/doc/spec/

[22] J. M. de Haro, R. Cano, C. Álvarez, D. Jiménez-González, X. Martorell, E. Ayguadé, J. Labarta, F. Abel, B. Ringlein, and B. Weiss, "OmpSs@cloudFPGA: An FPGA task-based programming model with message passing," in *2022 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2022, pp. 828–838.

[23] O. Shaaban, J. Aguilar, V. Beltran, P. Carpenter, E. Ayguadé, and J. L. Mancho, "Automatic aggregation of subtask accesses for nested OpenMP-style tasks," in *2022 IEEE 34th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*. IEEE, 2022, pp. 315–325. [Online]. Available: https://doi.org/10.1109/SBAC-PAD55451.2022.00042

[24] O. Shaaban, J. F. d'Albiat, I. Piedrahita, B. Vicenç, P. Carpenter, E. Ayguadé, and J. Labarta, "Leveraging iterative applications to improve the scalability of

task-based programming models on distributed systems," *Available at SSRN 4764389*, 2024.

[25] G. Taffoni *et al.*, "EuroEXA D2.6: Final ported application software," 2022. [Online]. Available: https://openaccess.inaf.it/bitstream/20.500.12386/33614/1/EUROEXA_D2.6_v1.1.6.pdf

[26] M. A. Pericàs *et al.*, "LEGaTO D3.4: "report on evaluation and optimizations in the runtime stack"," 2020. [Online]. Available: https://legato.bsc.es/sites/default/files/uploaded/d3.4.pdf

[27] J. M. Peréz Cáncer, "A dependency-aware parallel programming model," Ph.D. dissertation, Universitat Politècnica de Catalunya, 2015. [Online]. Available: http://dx.doi.org/10.5821/dissertation-2117-95636

[28] Barcelona Supercomputing Center. (2021) Influence in OpenMP - OmpSs-2 specification. [Online]. Available: https://pm.bsc.es/ftp/ompss-2/doc/spec/introduction/openmp.html

[29] D. Álvarez, K. Sala, M. Maroñas, A. Roca, and V. Beltran, *Advanced Synchronization Techniques for Task-Based Runtime Systems*. New York, NY, USA: Association for Computing Machinery, 2021, p. 334–347. [Online]. Available: https://doi.org/10.1145/3437801.3441601

[30] Barcelona Supercomputing Center, "OmpSs-2@Cluster releases," 2022. [Online]. Available: https://github.com/bsc-pm/ompss-2-cluster-releases

[31] ——. (2021) Nanos6. [Online]. Available: https://github.com/bsc-pm/nanos6

[32] ——. (2021) Mercurium. [Online]. Available: https://pm.bsc.es/mcxx

[33] ——. (2024) Extrae instrumentation package. [Online]. Available: https://github.com/bsc-performance-tools/extrae

[34] ——. (2024) Ompss-2 linter tool. [Online]. Available: https://github.com/bsc-pm/ompss-2-linter

[35] "mmap(2) Linux User's Manual." [Online]. Available: https://man7.org/linux/man-pages/man2/mmap.2.html

[36] J. M. Perez, R. M. Badia, and J. Labarta, "Handling task dependencies under strided and aliased references," in *Proceedings of the 24th ACM International Conference on Supercomputing*, 2010, pp. 263–274. [Online]. Available: https://doi.org/10.1145/1810085.1810122

[37] MPI Forum, "MPI documents." [Online]. Available: https://www.mpi-forum.org/docs/

[38] D. W. Walker and J. J. Dongarra, "Mpi: a standard message passing interface," *Supercomputer*, vol. 12, pp. 56–68, 1996.

[39] I. Laguna, R. Marshall, K. Mohror, M. Ruefenacht, A. Skjellum, and N. Sultana, "A large-scale study of mpi usage in open-source hpc applications," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2019, pp. 1–14.

[40] A. Hori, E. Jeannot, G. Bosilca, T. Ogura, B. Gerofi, J. Yin, and Y. Ishikawa, "An international survey on mpi users," *Parallel Computing*, vol. 108, p. 102853, 2021. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0167819121000983

[41] R. W. Numrich and J. Reid, "Co-array Fortran for parallel programming," in *ACM SIGPLAN Fortran Forum*, vol. 17, no. 2. ACM New York, NY, USA, 1998, pp. 1–31. [Online]. Available: https://doi.org/10.1145/289918.289920

[42] UPC Consortium, "Berkeley UPC – Unified Parallel C," 2024. [Online]. Available: https://upc.lbl.gov/

[43] J. Lee, M. T. Tran, T. Odajima, T. Boku, and M. Sato, "An extension of XcalableMP PGAS lanaguage for multi-node GPU clusters," in *Euro-Par 2011: Parallel Processing Workshops: CCPI, CGWS, HeteroPar, HiBB, HPCVirt, HPPC, HPSS, MDGS, ProPer, Resilience, UCHPC, VHPC, Bordeaux, France, August 29–September 2, 2011, Revised Selected Papers, Part I 17*. Springer, 2012, pp. 429–439. [Online]. Available: https://doi.org/10.1007/978-3-642-29737-3_48

[44] M. De Wael, S. Marr, B. De Fraine, T. Van Cutsem, and W. De Meuter, "Partitioned global address space languages," *ACM Comput. Surv.*, vol. 47, no. 4, may 2015. [Online]. Available: https://doi.org/10.1145/2716320

[45] D. Grünewald and C. Simmendinger, “The GASPI API specification and its implementation GPI 2.0,” in *7th International Conference on PGAS Programming Models*, vol. 243, 2013, p. 52.

[46] B. Chapman, T. Curtis, S. Pophale, S. Poole, J. Kuehn, C. Koelbel, and L. Smith, “Introducing OpenSHMEM: SHMEM for the PGAS community,” in *Proceedings of the Fourth Conference on Partitioned Global Address Space Programming Model*, 2010, pp. 1–3. [Online]. Available: https://doi.org/10.1145/2020373.2020375

[47] OpenACC Organization, “OpenACC: Directives for accelerators,” 2011. [Online]. Available: http://www.openacc-standard.org

[48] D. Luebke, “CUDA: Scalable parallel programming for high-performance scientific computing,” in *2008 5th IEEE international symposium on biomedical imaging: from nano to macro*. IEEE, 2008, pp. 836–838. [Online]. Available: https://doi.org/10.1109/ISBI.2008.4541126

[49] R. Rabenseifner and G. Wellein, “Comparison of parallel programming models on clusters of SMP nodes,” in *Modeling, Simulation and Optimization of Complex Processes: Proceedings of the International Conference on High Performance Scientific Computing, March 10–14, 2003, Hanoi, Vietnam*. Springer, 2005, pp. 409–425. [Online]. Available: https://doi.org/10.1007/3-540-27170-8_31

[50] G. Jost, H.-Q. Jin, F. F. Hatay *et al.*, “Comparing the OpenMP, MPI, and hybrid programming paradigm on an SMP cluster,” in *European Workshop on OpenMP and Applications 2003*, 2003. [Online]. Available: https://ntrs.nasa.gov/citations/20030107321

[51] M. McCool, J. Reinders, and A. Robison, *Structured Parallel Programming: Patterns for Efficient Computation*, 1st ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2012.

[52] D. Lea, “A java fork/join framework,” in *Proceedings of the ACM 2000 conference on Java Grande*, 2000, pp. 36–43.

[53] S. Chatterjee, S. Tasırlar, Z. Budimlic, V. Cavé, M. Chabbi, M. Grossman, V. Sarkar, and Y. Yan, “Integrating asynchronous task parallelism with MPI,” in *2013 IEEE 27th International Symposium on Parallel and Distributed Processing*. IEEE, 2013, pp. 712–725. [Online]. Available: https://doi.org/10.1109/IPDPS.2013.78

[54] K. Fürlinger, J. Gracia, A. Knüpfer, T. Fuchs, D. Hünich, P. Jungblut, R. Kowalewski, and J. Schuchart, "DASH: Distributed data structures and parallel algorithms in a global address space," in *Software for Exascale Computing-SPPEXA 2016-2019.* Springer International Publishing, 07 2020, pp. 103–142. [Online]. Available: https://doi.org/10.1007/978-3-030-47956-5_6

[55] G. Bosilca, A. Bouteiller, A. Danalis, T. Herault, P. Luszczek, and J. Dongarra, "Dense linear algebra on distributed heterogeneous hardware with a symbolic DAG approach," Lawrence Berkeley National Lab. (LBNL), Berkeley, CA (United States), Tech. Rep., 2012. [Online]. Available: https://www.osti.gov/servlets/purl/1173290

[56] P. Cardosi and B. Bramas, "Specx: a C++ task-based runtime system for heterogeneous distributed architectures," *arXiv preprint arXiv:2308.15964*, 2023.

[57] E. Agullo, O. Aumage, M. Faverge, N. Furmento, F. Pruvost, M. Sergent, and S. P. Thibault, "Achieving high performance on supercomputers with a sequential task-based programming model," *IEEE Transactions on Parallel and Distributed Systems*, 2017. [Online]. Available: https://doi.org/10.1109/TPDS.2017.2766064

[58] A. YarKhan, "Dynamic task execution on shared and distributed memory architectures," Ph.D. dissertation, University of Tennessee, 2012. [Online]. Available: https://trace.tennessee.edu/cgi/viewcontent.cgi?article=2774&context=utk_graddiss

[59] A. YarKhan, J. Kurzak, and J. Dongarra, "Quark users' guide: Queueing and runtime for kernels," Tech. Rep. ICL-UT-11-02, 2011-00 2011.

[60] F. Song and J. Dongarra, "A scalable framework for heterogeneous GPU-based clusters," in *Proceedings of the twenty-fourth annual ACM symposium on Parallelism in algorithms and architectures*, 2012, pp. 91–100.

[61] J. Bueno, J. Planas, A. Duran, R. M. Badia, X. Martorell, E. Ayguadé, and J. Labarta, "Productive programming of GPU clusters with OmpSs," in *IEEE 26th International Parallel and Distributed Processing Symposium*, 5 2012. [Online]. Available: https:doi.org//10.1109/IPDPS.2012.58

[62] M. Bauer, S. Treichler, E. Slaughter, and A. Aiken, "Legion: Expressing locality and independence with logical regions," in *SC '12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, 2012, pp. 1–11. [Online]. Available: https://doi.org/10.1109/SC.2012.71

[63] E. Slaughter, W. Lee, S. Treichler, M. Bauer, and A. Aiken, "Regent: a high-productivity programming language for HPC with logical regions," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2015, pp. 1–12. [Online]. Available: https://doi.org/10.1145/2807591.2807629

[64] Parallel Programming Lab, Dept of Computer Science, University of Illinois. (2023) Charm++ documentation. [Online]. Available: https://charm.readthedocs.io/en/latest/index.html

[65] H. Kaiser, T. Heller, B. Adelstein-Lelbach, A. Serio, and D. Fey, "HPX: A task based programming model in a global address space," in *8th International Conference on Partitioned Global Address Space Programming Models*, 2014. [Online]. Available: https://doi.org/10.13140/2.1.2635.5204

[66] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar, "X10: An object-oriented approach to non-uniform cluster computing," in *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, ser. OOPSLA '05. New York, NY, USA: Association for Computing Machinery, 2005, p. 519–538. [Online]. Available: https://doi.org/10.1145/1094811.1094852

[67] (2022) The chapel parallel programming language. [Online]. Available: https://chapel-lang.org/

[68] M. Weiland, "Chapel, fortress and x10: novel languages for hpc," *EPCC, The University of Edinburgh, Tech. Rep. HPCxTR0706*, vol. 1, 2007.

[69] C. Pheatt, "Intel® threading building blocks," *Journal of Computing Sciences in Colleges*, vol. 23, no. 4, pp. 298–298, 2008.

[70] M. Learn, "Task parallel library (tpl) - .net," *Microsoft Learn*, 2022. [Online]. Available: https://learn.microsoft.com/en-us/dotnet/standard/parallel-programming/task-parallel-library-tpl

[71] Google, "Tensorflow," *TensorFlow*, 2022. [Online]. Available: https://www.tensorflow.org/

[72] F. Lordan, E. Tejedor, J. Ejarque, R. Rafanell, J. Alvarez, F. Marozzo, D. Lezzi, R. Sirvent, D. Talia, and R. M. Badia, "ServiceSs: An interoperable programming

framework for the cloud," *Journal of grid computing*, vol. 12, no. 1, 2014. [Online]. Available: https://doi.org/10.1007/s10723-013-9272-5

[73] E. Deelman, G. Singh, M.-H. Su, J. Blythe, Y. Gil, C. Kesselman, G. Mehta, K. Vahi, G. Berriman, J. Good, A. Laity, and D. S. Katz, "Pegasus: A framework for mapping complex scientific workflows onto distributed systems," *Scientific Programming*, vol. 13, no. 3, pp. 219–237, Jan 2005. [Online]. Available: https://doi.org/10.1155/2005/128026

[74] T. Rotaru, M. Rahn, and F.-J. Pfreundt, "MapReduce in GPI-Space," in *Euro-Par 2013: Parallel Processing Workshops.* Berlin, Heidelberg: Springer Berlin Heidelberg, 2014, pp. 43–52. [Online]. Available: https://doi.org/10.1007/978-3-642-54420-0_5

[75] H. Bae, D. Mustafa, J.-W. Lee, H. Lin, C. Dave, R. Eigenmann, S. P. Midkiff *et al.*, "The Cetus source-to-source compiler infrastructure: overview and evaluation," *International Journal of Parallel Programming*, vol. 41, no. 6, pp. 753–767, 2013.

[76] G. Souza Diniz Mendonça, B. Campos Ferreira Guimarães, P. R. Oliveira Alves, F. M. Quintão Pereira, M. M. Pereira, and G. Araújo, "Automatic insertion of copy annotation in data-parallel programs," in *2016 28th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, 2016, pp. 34–41.

[77] G. Mendonça, B. Guimarães, P. Alves, M. Pereira, G. Araújo, and F. M. Q. Pereira, "DawnCC: automatic annotation for data parallelism and offloading," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 14, no. 2, pp. 1–25, 2017.

[78] C. Liao, D. J. Quinlan, J. J. Willcock, and T. Panas, "Semantic-aware automatic parallelization of modern applications using high-level abstractions," *International Journal of Parallel Programming*, vol. 38, no. 5, pp. 361–378, 2010.

[79] U. Bondhugula, A. Hartono, J. Ramanujam, and P. Sadayappan, "A practical automatic polyhedral parallelizer and locality optimizer," in *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2008, pp. 101–113.

[80] P. Ramos, G. Souza, D. Soares, G. Araújo, and F. M. Q. Pereira, "Automatic annotation of tasks in structured code," in *Proceedings of the 27th International Conference on Parallel Architectures and Compilation Techniques*, 2018, pp. 1–13.

[81] E. Ayguadé, R. M. Badia, D. Jiménez, J. R. Herrero, J. Labarta, V. Subotic, and G. Utrera, "Tareador: a tool to unveil parallelization strategies at undergraduate level," in *Proceedings of the Workshop on Computer Architecture Education*, 2015, pp. 1–8.

[82] P. M. Carpenter, A. Ramirez, and E. Ayguadé, "Starsscheck: A tool to find errors in task-based parallel programs," in *European Conference on Parallel Processing*. Springer, 2010, pp. 2–13.

[83] J. Seward *et al.* (2023) Valgrind. [Online]. Available: https://valgrind.org

[84] S. Economo, S. Royuela, E. Ayguadé, and V. Beltran, "A toolchain to verify the parallelization of OmpSs-2 applications," in *Euro-Par 2020: Parallel Processing*, M. Malawski and K. Rzadca, Eds. Cham: Springer International Publishing, 2020, pp. 18–33.

[85] V. Pillet, J. Labarta, T. Cortes, and S. Girona, "Paraver: A tool to visualize and analyze parallel code," in *Proceedings of WoTUG-18: transputer and occam developments*, vol. 44, no. 1, 1995, pp. 17–31.

[86] S. MD. Slurm workload manager. [Online]. Available: https://slurm.schedmd.com/overview.html

[87] Barcelona Supercomputing Center, "MareNostrum 4 (2017) System Architecture," https://www.bsc.es/marenostrum/marenostrum/technical-information, 2017. [Online]. Available: https://www.bsc.es/marenostrum/marenostrum/technical-information

[88] ——. (2022) OmpSs-2 examples. [Online]. Available: https://pm.bsc.es/gitlab/ompss-2/examples

[89] P. Barkman, "Parallel Barnes–Hut algorithm," https://github.com/barkm/n-body, 2019.

[90] J. K. Salmon, "Parallel hierarchical n-body methods," Ph.D. dissertation, California Institute of Technology, 1991.

[91] S. Yu and S. Zhou, "A survey on metric of software complexity," in *2010 2nd IEEE International conference on information management and engineering*. IEEE, 2010, pp. 352–356.

[92] R. Vidal, M. Casas, M. Moretó, D. Chasapis, R. Ferrer, X. Martorell, E. Ayguadé, J. Labarta, and M. Valero, “Evaluating the impact of OpenMP 4.0 extensions on relevant parallel workloads,” in *OpenMP: Heterogenous Execution and Data Movements: 11th International Workshop on OpenMP, IWOMP 2015, Aachen, Germany, October 1-2, 2015, Proceedings 11.* Springer, 2015, pp. 60–72. [Online]. Available: http://dx.doi.org/10.1007/978-3-319-24595-9_5

[93] A. Navarro, A. F. Lorenzon, E. Ayguadé, and V. Beltran, “Enhancing resource management through prediction-based policies,” in *European Conference on Parallel Processing.* Springer, 2020, pp. 493–509. [Online]. Available: https://doi.org/10.1007/978-3-030-57675-2_31